ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Generador de casos de prueba genético

Álvaro Galán Piñero

17 de septiembre de 2012

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE SISTEMAS

Generador de casos de prueba genético

- Departamento: Ingeniería Informática.
- Directores del proyecto: Juan José Domínguez Jiménez, Antonio García Domínguez.
- Autor del proyecto: Álvaro Galán Piñero.

Cádiz, 17 de septiembre de 2012

Fdo: Álvaro Galán Piñero

Licencia

Notación y formato

En esta sección, incluiremos los aspectos relevantes a la notación y el formato a lo largo del documento. Dicha notación es la siguiente:

Si nos vamos a referir a un directorio en particular, usaremos la notación:

/home

Cuando nos refiramos a un programa en concreto, utilizaremos la notación:

emacs.

Cuando nos refiramos a un comando, o función de un lenguaje, usaremos la notación:

`quicksort`.

Si nos vamos a referir a una clase, utilizaremos la notación:

MiClase

Agradecimientos

- A mi familia y amigos, por apoyarme y animarme en los meses de duro trabajo.
- A Inmaculada Medina Bulo, por permitirme colaborar en el grupo UCASE mediante la realización de este proyecto.
- A Antonio García Domínguez y Juan José Domínguez Jiménez, por ser mis tutores del proyecto y resolver las dudas planteadas.
- Al resto del grupo UCASE, por su apoyo y ayuda a lo largo del proyecto.

ÍNDICE GENERAL

INDICE DE FIGURAS

INDICE DE LISTADOS

CAPÍTULO 1

MOTIVACIÓN Y CONTEXTO

1.1. Introducción

Los algoritmos genéticos (AG) son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Se llaman así porque se inspiran en la evolución biológica y su base genético-molecular. A lo largo de las generaciones, las poblaciones evolucionan siguiendo los principios de la selección natural y la supervivencia de los más fuertes.

Los individuos con un mayor éxito en la supervivencia y perpetuación de la especie tienen mayor probabilidad de generar un gran número de supervivientes. En cambio, los que peor se adapten tendrán una probabilidad menor. Todo esto conlleva que los genes de los individuos que mejor se hayan adaptados se irán propagando a lo largo de las sucesivas generaciones mientras que los genes de los peor adaptados se irán perdiendo. De esta forma, las especies evolucionan logrando individuos cuyas características están mejor adaptadas al medio en el que viven, llegando incluso a producir superindividuos.

Los algoritmos genéticos usan una analogía directa con el comportamiento natural. Trabajan con una población de individuos, cada uno de los cuales tienen asignado un valor o aptitud que nos informa de cuán bueno es un individuo con respecto a otro. Cuan-

ta mayor sea la adaptación del individuo, mayor será la probabilidad de supervivencia de dicha especie, dicho valor o aptitud también será mayor y por tanto, aumentará la probabilidad de reproducción con individuos de la misma especie.

El presente Proyecto Fin de Carrera se enmarca como trabajo de colaboración dentro del grupo UCASE de Ingeniería del Software. Dentro de la línea de pruebas del software se trabaja en la mejora de conjuntos de casos de prueba para composiciones de servicios Web escritas en WS-BPEL.

El lenguaje WS-BPEL es, como se ha comentado, un lenguaje para la composición de servicios web basado en XML. Se caracteriza porque permite desarrollar procesos de negocio a partir de Servicios Web (WS) preexistentes y ofrecerlos como otro WS.

En un Proyecto Fin de Carrera anterior, "Analizador de Servicios Web basados en WSDL 1.1 para pruebas paramétricas", se implementó *ServiceAnalyzer* [1], un analizador de servicios Web, que generaba plantillas parametrizadas para producir mensajes de acuerdo a todas las restricciones impuestas desde WSDL [2], XML Schema [3] y el WS-I Basic Profile 1.1 [4]. Con eso se tenía parte del trabajo realizado, pero ahora faltaba otra cuestión: producir los datos con que se rellenarán esas plantillas.

Posteriormente, en otro Proyecto Fin de Carrera, "Generador de casos de prueba aleatorio basado en especificaciones abstractas", se implementó *TestGenerator* [5], una aplicación que generaba los datos necesarios de forma aleatoria con unas restricciones específicas dadas en un fichero de entrada al programa.

En este Proyecto Fin de Carrera: se implementará *GameraHOM-ggen*, una aplicación que a partir de los datos generados de forma aleatoria, generará otros nuevos mediante un algoritmo genético. De esta forma, los casos de prueba obtenidos serán más concretos y robustos para detectar posibles fallos.

1.2. Objetivos

El proyecto pretende satisfacer los siguientes objetivos:

- Deberá ser capaz de trabajar con datos de diferentes tipos, los usados para las

diversas composiciones, tales como: enteros, flotantes, cadenas, listas...

- Los datos generados finalmente tras aplicar el algoritmo genético deberán presentarse en un formato que sea utilizable por las herramientas del grupo de investigación. Deberán ser compatibles con BPELUnit, el marco de pruebas unitarias para WS-BPEL que se emplea en el grupo.

- Crear un fichero en el cual se configurarán los parámetros de entrada del algoritmo. De esta forma se podrá fácilmente cambiar la configuración del mismo para posteriores estudios que se quieran realizar.

- Codificar los casos de prueba adecuadamente. En algoritmos genéticos se habla de individuos y población. Un caso de prueba será un individuo y habrá que definir su estructura de forma que se pueda trabajar fácilmente con ellos.

- Implementar los operadores genéticos necesarios, mutación y cruce, los de selección y un generador de nuevos individuos.

- Implementar los diferentes criterios de parada del algoritmo.

1.3. Alcance

Este Proyecto Fin de Carrera trabajará con el sistema de tipos de las composiciones WS-BPEL que actualmente se están estudiando, de forma que cubre las necesidades del grupo UCASE existentes por el momento.

Por limitaciones de tiempo, el presente trabajo únicamente recoge el desarrollo del algoritmo genético, dejando a un lado estudios estadísticos que nos pudiesen informar por ejemplo, de la calidad de los casos de prueba generados.

La comunicación entre el usuario y la herramienta se hará a través de la línea de comandos.

1.4. Conceptos básicos

Se pasarán a explicar a continuación algunos de los conceptos básicos necesarios para entender como encaja este proyecto en el grupo de investigación UCASE. Las tecnologías de las que de alguna manera, ya sea directa o indirectamente, afectan a los dominios de aplicación de la herramienta a desarrollar.

1.4.1. WS-BPEL

WS-BPEL [6] (Business Process Execution Language o Lenguaje de Ejecución de Procesos de Negocios con Servicios Web en castellano), es el lenguaje usado en algunas de las líneas de trabajo del grupo UCASE.

WS-BPEL es un lenguaje estandarizado por OASIS para la composición de servicios web. Está basado en XML y diseñado para el control centralizado de la invocación de diferentes servicios Web, con cierta lógica de negocio añadida que ayuda a la programación en gran escala.

La estructura de un proceso WS-BPEL se divide en cuatro secciones:

1. La definición de relaciones con los socios externos (partnerLinks), el cliente que utiliza el proceso de negocio y los WS a los que llama el proceso.

2. La definición de las variables que emplea el proceso.

3. La definición de los distintos tipos de manejadores que puede utilizar un proceso, tales como manejadores de fallos y eventos. Los manejadores de fallos indican las acciones a realizar en caso de producirse un fallo interno o en un WS al que se llama. Los manejadores de eventos especifican las acciones a realizar en caso de que el proceso reciba una petición durante su flujo normal de ejecución.

4. Por último, se decribe el comportamiento del proceso de negocio; esto se logra a través de las actividades que proporciona el lenguaje.

Todos los elementos declarados dentro del proceso son globales. Podemos también

declalarlos de forma local mediante el contenedor scope, que permite dividir el proceso de negocio en diferentes ámbitos.

Los principales elementos constructivos son las actividades. Éstas pueden ser de dos tipos: básicas y estructuradas. Las actividades básicas son las que realizan una labor determinada (recibir un mensaje, manipular datos, etc.), mientras que las actividades estructuradas pueden contener otras actividades y definen la lógica de negocio.

A las actividades pueden asociarse un conjunto de atributos y un conjunto de contenedores, que pueden incluir diferentes elementos que a su vez pueden tener atributos asociados. En el listado 1.1 tenemos un ejemplo.

Listado 1.1: Fragmento de una composición WS-BPEL

```
<flow>
 <links>
  <link name="comprobarVuelo-reservarVuelo"
 </links>
 <invoke name="comprobarVuelo" . . . >
  <sources>
   <source linkName="comprobarVuelo-reservarVuelo"
    </sources>
 </invoke>
 <invoke name="comprobarHotel" . . . />
 <invoke name="comprobarAlquilerCoche" . . . />
 <invoke name="reservarVuelo" . . . >
 <targets>
  <target linkName="comprobarVuelo-reservarVuelo" />
 </targets>
 </invoke>
</flow>
```

En dicho ejemplo podemos observar que una actividad estructurada ha sido creada, con la etiqueta “flow” y como podemos establecer una sincronización entre algunos actividades. En el caso que nos ocupa, podemos intuir que no tienen sentido ejecutar la

acción “reservarVuelo” antes de “comprobarVuelo”. Por ello, creamos un enlace entre ambas actividades (líneas 2 a 4). Si seguimos leyendo, observamos que la primera actividad que lanzamos será “comprobarVuelo”, y en las líneas siguientes (líneas 5 a 9) la establecemos como un comienzo del enlace. A continuación, vamos lanzando el resto de actividades una a una(líneas 10 a 12).

1.4.2. SOAP

SOAP [7] (Simple Object Access Protocol) es un protocolo estándar que define como dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. Este protocolo deriva de un protocolo creado por David Winer en 1998, llamado XML-RPC. SOAP fue creado por Microsoft, IMB y otros y está actualmente bajo el auspicio de la W3C. Es uno de los protocolos utilizados en los servicios Web. En la figura 1.1 podemos observar la estructura básica de un mensaje SOAP.

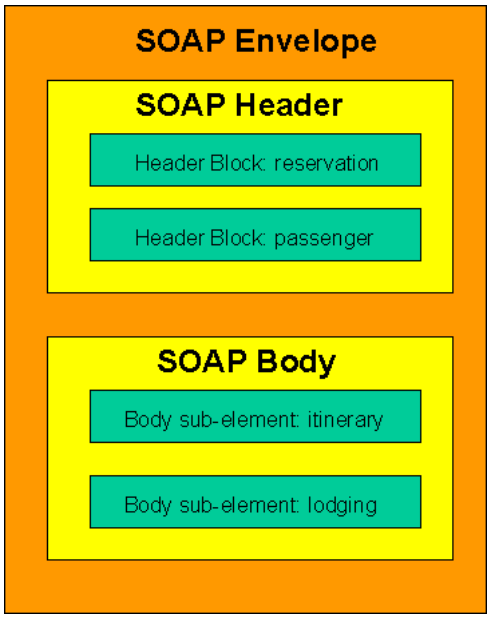

Figura 1.1.: Estructura mensaje SOAP

1.4.3. XML Schema

XML Schema es un lenguaje utilizado para describir la estructura y las restricciones de los contenidos de los documentos XML. De esta forma podemos controlar la estructura y los tipos de datos de una manera muy amplia. Podemos ver un ejemplo en el listado 1.2 en el que tenemos un libro definido por cuatro elementos: título, autores, edición y número de páginas, todos ellos de tipo cadena.

Listado 1.2: Ejemplo de XML Schema

```
1  <?xml version="1.0"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <xs:element name="libro">
5    <xs:complexType>
6      <xs:sequence>
7        <xs:element name="Título" type="xs:string"/>
8        <xs:element name="Autores" type="xs:string"/>
9        <xs:element name="Editorial" type="xs:string"/>
10       <xs:element name="Páginas" type="xs:string"/>
11     </xs:sequence>
12   </xs:complexType>
13 </xs:element>
14
15 </xs:schema>
```

1.4.4. BPELUnit

BPELUnit [8] es un marco de pruebas unitarias para composiciones WS-BPEL. Provee facilidades para la implementación, la invocación y la anulación de la implementación de una composición que funcione con un motor WS-BPEL. Los servicios externos se pueden sustituir por «mockups» y las respuestas se pueden retrasar a voluntad. Además, BPELUnit pueden crear casos de prueba mediante la lectura de un fichero de datos

externo y combinarlo con las plantillas de Apache Velocity.

En el caso del presente proyecto, se utilizan BPTS basados en plantillas. De esta forma se construyen el cuerpo de los mensajes SOAP para ser enviados desde los «mockups» utilizando plantillas Velocity. Estas plantillas permiten al usuario definir fácilmente muchos casos de prueba que tienen las mismas actividades, pero tienen un contenido diferente en sus mensajes. De esta forma se facilita la generación de casos de prueba y su automatización.

En el listado 1.3 podemos ver un ejemplo de un de un fichero BPTS basado en plantilla.

Listado 1.3: Ejemplo de un fichero BPTS basado en plantilla

```
<?xml version="1.0" encoding="UTF-8"?>
<tes:testSuite
  xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
  xmlns:as="http://j2ee.netbeans.org/wsdl/ConcreteAssessorService"
  xmlns:sp="http://j2ee.netbeans.org/wsdl/ConcreteLoanService"
  xmlns:gen="http://j2ee.netbeans.org/wsdl/loanServicePT"
  xmlns:pr="http://enterprise.netbeans.org/bpel/LoanApproval_V2/
  loanApprovalProcess"
  xmlns:tes="http://www.bpelunit.org/schema/testSuite">

 <tes:name>loanApprovalProcess</tes:name>
 <tes:baseURL>http://localhost:7777/ws</tes:baseURL>

 <tes:deployment>
   <tes:put name="loanApprovalProcess" type="activebpel">
     <tes:wsdl>LoanService.wsdl</tes:wsdl>
     <tes:property name="BPRFile">LoanApprovalRPC.bpr</tes:property>
   </tes:put>
   <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
   <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
 </tes:deployment>

```

```
23 <tes:testCases>
24   <tes:testCase name="MainTemplate" basedOn="" abstract="false"
25    vary="false">
26     <tes:setUp>
27       <tes:dataSource type="velocity" src="data.vm">
28         <tes:property name="iteratedVars">firstName surName cantidad
29          accepted riskLevel</tes:property>
30       </tes:dataSource>
31     </tes:setUp>
32     <tes:clientTrack>
33       <tes:sendReceive
34          service="sp:LoanService"
35          port="LoanServicePort"
36          operation="request">
37
38         <tes:send fault="false">
39           <tes:template>
40             <gen:name>$firstName $surName</gen:name>
41             <gen:firstName>$firstName</gen:firstName>
42             <gen:cantidad>$cantidad</gen:cantidad>
43           </tes:template>
44         </tes:send>
45
46         <tes:receive fault="false">
47           <tes:condition>
48             <tes:expression>accept</tes:expression>
49             <tes:value>$accepted</tes:value>
50           </tes:condition>
51         </tes:receive>
52       </tes:sendReceive>
53     </tes:clientTrack>
54
55     <tes:partnerTrack
```

```
56        name="assessor"
57        assume="10000 > $cantidad">
58        <tes:receiveSend
59           service="as:RiskAssessmentService"
60           port="RiskAssessmentPort"
61           operation="check">
62          <tes:receive fault="false"/>
63          <tes:send fault="false">
64            <tes:template>
65              <gen:riskLevel>$riskLevel</gen:riskLevel>
66            </tes:template>
67          </tes:send>
68        </tes:receiveSend>
69      </tes:partnerTrack>
70
71      <tes:partnerTrack
72        name="approver"
73        assume="$riskLevel = 'high' or $cantidad>=10000">
74        <tes:receiveSend
75           service="ap:ApprovalService"
76           port="ApprovalServicePort"
77           operation="approve">
78          <tes:receive fault="false"/>
79          <tes:send fault="false">
80            <tes:template>
81              <gen:accept>$accepted</gen:accept>
82            </tes:template>
83          </tes:send>
84        </tes:receiveSend>
85      </tes:partnerTrack>
86    </tes:testCase>
87  </tes:testCases>
88 </tes:testSuite>
```

El listado 1.3 se corresponde con un fragmento de la clásica composición WS-BPEL «LoanApproval». Dicha composición explica el procedimiento a seguir para aprobar o no un préstamo solicitado por un cliente.

Hasta la línea 8 tenemos el elemento raíz y la definición de los espacios de nombres XML. El prefijo `tes` está asociado al espacio de nombres de BPELUnit, y el resto son prefijos propios de esta composición y de los envoltorios SOAP.

Posteriormente, en la sección de despliegue (líneas 10-21) podemos ver que la composición se comunica con dos «mockups»: el asesor y el aprobador.

La línea 22 en adelante, se corresponde con la sección de los casos de prueba. Cada caso de prueba lo extraeremos del fichero de datos externo (cargado en la línea 27). Dicho fichero estará en formato velocity, que posteriormente explicaremos (ver 1.4.5), en él se encuentran las variables que aparecen a continuación: el nombre y apellidos del cliente, la cantidad pedida, si el préstamo es o no es aceptado y el riesgo que posee el cliente respectivamente.

Posteriormente, se envían los datos de cada uno de los clientes y si el préstamo será o no aceptado.

A partir de la línea 55 podemos observar que si la cantidad es menor que 10000, el proceso WS-BPEL invoca al WS asesor («asessor») o al aprobador («approver») en caso contrario. Cuando llamamos al asesor, la salida se corresponde con el riesgo que posee el cliente, si el riesgo es bajo se acepta el préstamo y si es alto se llama al aprobador.

Si el riesgo es alto y la cantidad mayor o igual que 10000, el préstamo será aceptado o no en función del valor que haya tomado la variable «accepted».

1.4.5. Apache Velocity

Apache Velocity [9] es un motor de plantillas basado en Java. Le permite a los diseñadores de páginas hacer referencia a métodos definidos dentro del código Java. Los diseñadores Web pueden trabajar en paralelo con los programadores Java para desarrollar sitios de acuerdo al modelo de Modelo-Vista-Controlador (MVC), permitiendo que los diseñadores se concentren únicamente en crear un sitio bien diseñado y que los

programadores se encarguen solamente de escribir código de primera calidad. Velocity separa el código Java de las páginas Web, haciendo el sitio más mantenible a largo plazo y presentando una alternativa viable a Java Server Pages (JSP) o PHP.

A continuación, en el listado 1.4 podemos ver un ejemplo muy sencillo de como funciona Velocity. Vamos a crear una página web en la que únicamente se visualice el mensaje "Hola mundo Velocity". Para ello, asignaremos en la línea 3 a la variable `foo` el valor ´´Velocity" y, posteriormente, imprimiremos el mensaje deseado haciendo referecia a dicha variable.

Listado 1.4: Ejemplo de una plantilla Velocity

```
1 <html>
2       <body>
3             #set( foo = "Velocity" )
4             Hola Mundo foo
5       </body>
6 <html>
```

En el caso que nos ocupa, usaremos los ficheros Velocity para definir el valor que tomen los casos de prueba en cada una de las generaciones. El fichero BPTS cargará el Velocity para tomar de él, como hemos comentado, el valor de cada una de las variables.

Cada fila del fichero Velocity comienza de la forma `#set`, seguido del nombre de la variable correspondiente y a continuación los distintos valores que toma esa variable para cada uno de los casos de prueba. Cada caso de prueba se corresponde con una columna en particular.

Es decir, para el listado 1.5 y siguiendo con el ejemplo de la composición «LoanApproval» el primer caso de prueba se correspondería con Juan Perez, pidiendo un préstamo por una cantidad con valor 486078, resultando el préstamo aceptado y teniendo el cliente un riesgo alto.

Listado 1.5: Salida en formato Velocity

```
1 #set($riskLevel = ["high", "high", "high", "low", "low"])
2 #set($accepted = ["true", "true", "true", "true", "false"])
```

```
3 #set($cantidad = [486078, 15885, 391253, 865931, 378163])
4 #set($surName = ["Perez", "Gomez", "Sanchez", "Gomez", "Sanchez"])
5 #set($firstName = ["Juan", "Antonio", "Manuel", "Juan", "Maria"])
```

1.4.6. Prueba de mutaciones

La prueba de mutaciones es una técnica de pruebas basada en introducir pequeños fallos sintácticos en el programa original. Como resultado, se generan unos programas, llamados mutantes, iguales al original salvo por la diferencia introducida.

El objetivo que persigue consiste en medir la calidad de un conjunto de casos de prueba.

Los mutantes son generados aplicando al programa original un conjunto de reglas, llamados operadores de mutación. Los operadores de mutación representan los errores que los programadores pueden y suelen cometer en algunas ocasiones.

A continuación mostraremos un ejemplo clarificante. Supongamos la siguiente operación lógica, en la que comprobamos si el precio de un determinado artículo es mayor que 10000.

$precio > 10000$

Ahora, la instrucción ha sufrido una mutación, pues el operador “>” pasa a ser “<”.

$precio < 10000$

Dependiendo del número de cambios realizados sobre el programa original hablamos de distintos órdenes de mutación. La mutación de orden 1 es aquella que resulta tras aplicarle un sólo cambio al programa original, de orden 2 se le aplicarán 2 cambios, 3 cambios a la de orden 3, y así sucesivamente...

Una vez que tenemos el código mutado, se ejecuta tanto el código original como el código mutado frente a un conjunto de casos de prueba. Cuando el código mutado y el código original dan el mismo resultado para todas las pruebas realizadas, decimos que el mutante permanece vivo. En caso de que para alguno de los casos de prueba las salidas sean distintas, decimos que el mutante está muerto. Si el código mutado provoca un error en la ejecución, decimos que el mutante es erróneo.

Uno de los problemas que existe en la prueba de mutaciones es la existencia de mutantes equivalentes. Se les llama así a los mutantes que siempre producen la misma salida que el programa original, es decir, se comportan de igual forma. No hay que confundir los mutantes equivalentes con los resistentes ya que los mutantes resistentes son aquellos producidos porque el caso de prueba no es lo suficientemente bueno para detectar el cambio.

En el siguiente diagrama 1.2 podemos observar que tras aplicar los diferentes casos de prueba tanto sobre el código original como sobre el código mutado nos encontramos con los 3 tipos de mutantes comentados anteriormente. Dependiendo de si las salidas son iguales, distintas o se ha producido un error en la ejecución, hablamos de mutantes vivos, muertos o erróneos respectivamente.

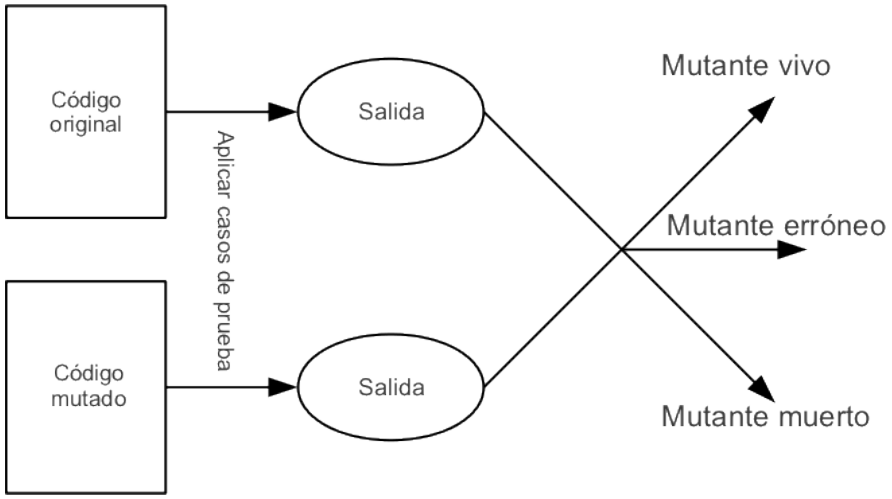

Figura 1.2.: Diagrama posibles salidas en las pruebas de mutaciones

Fases

Hasta llegar a poder comprobar si el mutante está vivo, muerto o es erróneo pasamos por diversas fases.

Primero, en una etapa de **análisis** se identifican los posibles cambios que puede sufrir el código BPEL del programa original.

Luego, en la fase de **mutación** se generan los diferentes mutantes. Las instrucciones que se hayan cambiado en cada mutante con respecto al programa original vendrán condicionadas por la fase anterior, donde fueron identificadas. Depediendo del número de cambios que se hayan realizado con respecto al programa original hablaremos de mutación de orden 1, 2, 3...

Por último, en la fase de **ejecución**, se ejecutan tanto el programa original como cada uno de los ficheros mutados contra un conjunto de casos de pruebas. Es en esta fase donde comparando las salidas del programa original y de los diversos mutantes determinamos si el mutante se encuentra vivo, muerto o es erróneo, marcando cada una de las pruebas con los valores 0, 1 ó 2 respectivamente.

Por tanto tendremos una matriz de salida, llamada matriz de ejecución, en la que podremos ver para cada mutante y cada prueba, el resultado obtenido.

CAPÍTULO 2

PLANIFICACIÓN

En este capítulo hablaremos de cómo se ha organizado el proyecto en el tiempo y de la metodología elegida para la realización del mismo. Describiremos cuáles han sido las principales etapas en las que podríamos descomponerlo y las características de cada una de ellas. Mostraremos también un diagrama de Gantt, para poder ver la cronología del proyecto con el paso del tiempo y extraer a través de él, las actividades más costosas.

El proyecto se ha desarrollado a lo largo de unos 10 meses. Durante los meses de noviembre-diciembre de 2011 se inició el trabajo, con el estudio de las tecnologías, hasta los meses de agosto-septiembre de 2012. Durante los meses de verano, Julio y Agosto, es cuando se han dedicado más horas al mismo, y durante los cuales se ha podido terminar de realizar el trabajo propuesto.

En un primer momento, hablando con el profesor Francisco Palomo Lozano sobre las diversas alternativas que existían para realizar mi Proyecto Fin de Carrera, me invitó a asistir a los seminarios del grupo de investigación UCASE, que realizan uno cada semana. Una vez allí, la profesora Inmaculada Medina Bulo, coordinadora del grupo de investigación, me puso en contacto con Juan José Domínguez Jiménez para implementar un generador de casos de prueba genético, herramienta que estaba pendiente de ser desarrollada en el mismo. Acepté la propuesta ante la idea de realizar un tipo de trabajo

totalmente distinto al realizado durante los 3 años de carrera anteriores y con el que sin duda aprendería mucho. Posteriormente, a principios del mes de febrero de 2012 y en una reunión con los profesores Juan José Domínguez Jiménez y Antonio García Domínguez, directores del proyecto, me explicaron en qué consistiría el trabajo a realizar.

2.1. Metodología

La metodología usada se corresponde con una metodología iterativa basada en prototipos siguiendo el modelo de ciclo de vida incremental. En la figura 2.1 podemos ver las principales características de este modelo. Está basado en el modelo lineal secuencial, pero se le añade cierta evolución al sistema, gracias a que las fases de análisis, diseño, implementación y prueba no sólo ocurren una vez.

Las características más importantes que podemos mencionar son: se parte de unos requisitos iniciales, y tras cada iteración se añaden nuevas funcionalidades al sistema obteniendo un producto software que puede ser utilizado. No es necesario esperar hasta el final del proceso para obtener un producto software, al contrario que en el modelo lineal secuencial. Otra ventaja que posee esta metodología es que no se hace necesario que todos los requisitos estén definidos al comienzo, si no que tras cada iteración, éstos pueden redefinirse.

Esta metodología se adapta perfectamente a nuestras necesidades ya que, al no saber hasta qué punto y cómo se puede realizar el diseño del problema, se hace necesario ir cumpliendo pequeños hitos e ir ampliando los requisitos en cada fase hasta completar el producto final. Además, tiene la ventaja de poder realizar pruebas al final de cada fase.

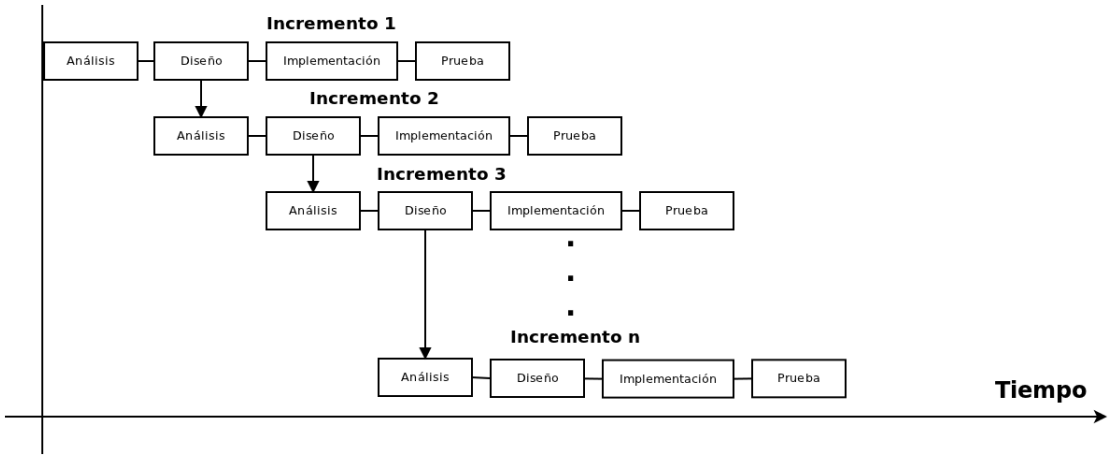

Figura 2.1.: Modelo de ciclo de vida incremental

2.2. Etapas

A continuación se explicarán las diferentes etapas por las que ha pasado el proyecto durante su desarrollo.

2.2.1. Estudio de las tecnologías y dominio del problema

Esta primera fase supone una toma de contacto tanto con el dominio del problema, los algoritmos genéticos, así como con el estudio de las tecnologías a utilizar. Ambas cosas eran pŕacticamente desconocidas antes de empezar a trabajar en el proyecto, por lo que si bien se ha extendendido durante todo el tiempo que éste ha durado, la dedicación a esta parte fue bastante costosa en los primeros meses de trabajo.

1. En un primer momento, se buscó información sobre el funcionamiento de los algoritmos genéticos [10], ya que durante los 3 años de carrera anteriores este tipo de algoritmos no aparecen.

2. Una vez comprendido este punto, se empiezan a leer artículos y otros Proyectos Fin de Carrera realizados por otros miembros del grupo de investigación ya que,

en algunos casos de manera directa y en otros indirecta, tenían algo que ver para la realización de éste.

3. Un porcentaje de esfuerzo amplio consistió en comprender las diversas tecnologías necesarias para la realización de este proyecto, en su mayoría prácticamente desde cero. Entre las tecnologías usadas cabe destacar WS-BPEL, BPELUnit, Java[11], el lenguaje de programacíon usado, JUnit, un framework para la automatización de las pruebas, YAML, SVN o Maven.

2.2.2. Elicitación de requisitos

Los requisitos se analizaron a través de varias reuniones con los miembros del grupo de investigación, tanto presencialmente como por correo electrónico. La mayoría de los requisitos se establecieron al principio del proyecto, pero otros como por ejemplo las condiciones de terminación del algoritmo se establecieron más tarde.

2.2.3. Configuración de los parámetros de entrada

Los parámetros de entrada del algoritmo, para que éste sea fácilmente configurable y para facilitar estudios posteriores estadísticos pudiendo cambiar las condiciones de partida, se dan escribiéndolos en un fichero YAML [12] que usando introspección se depositarán en una clase Java.

2.2.4. Algoritmo principal

Esta fase, el corazón de la aplicación, es donde se implementan los pasos a seguir por el algoritmo genético durante su ejecución. Lo primero que se hace indispensable es definir la estructura de los nuevos individuos de *GAmeraHOM-ggen*, para a partir de ahí definir el resto de operadores y operaciones a realizar.

2.2.5. Implementación de los operadores

En esta fase se implementan los operadores genéticos, como son la mutación y el cruce, los operadores de selección o los generadores de nuevos individuos. También las condiciones que deben cumplirse para dar por concluida la ejecución del algoritmo.

2.2.6. Pruebas

Esta etapa es muy importante ya que, como todo proyecto, este también tiene como objetivo producir software de la mejor calidad posible.

EL framework usado para la realización de las mismas es JUnit [13]. Cada parte del código lleva asociada una serie de pruebas que nos hacen comprobar tanto que nuestra aplicación hace lo que tiene que hacer como que no hace lo que no debe hacer. Buscamos posibles errores e intentamos subsanarlos.

2.2.7. Documentación

La memoria, si bien se ha intensificado su desarrollo durante los últimos meses de trabajo, se ha intentado llevar a la par durante la implementación de la aplicación en la medida de lo posible.

2.3. Diagrama Gantt

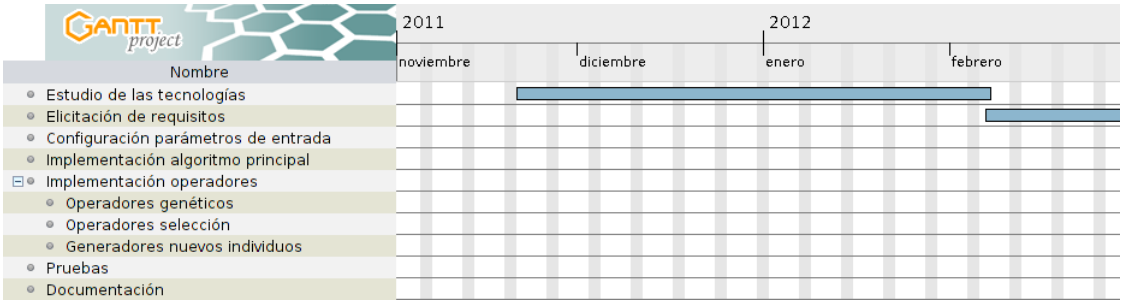

Figura 2.2.: Diagrama de Gantt I

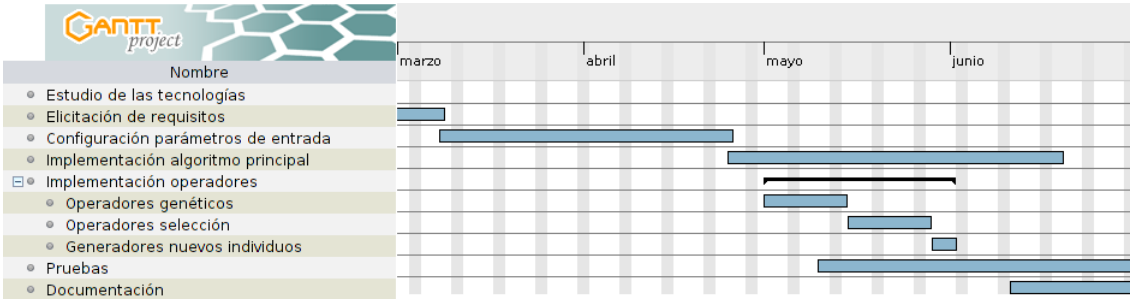

Figura 2.3.: Diagrama de Gantt II

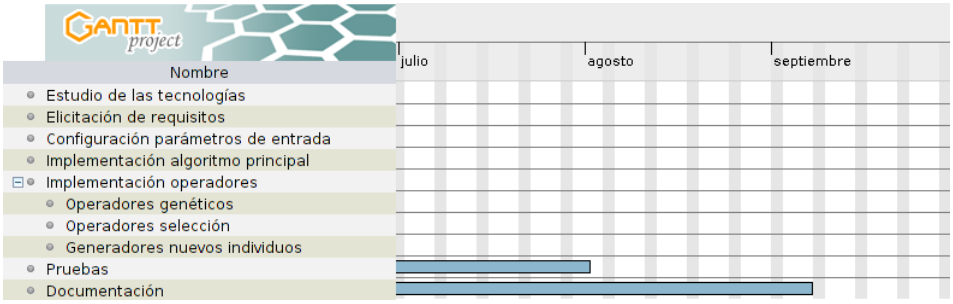

Figura 2.4.: Diagrama de Gantt III

CAPÍTULO 3

ALGORITMOS GENÉTICOS

En este capítulo se presentarán los algoritmos genéticos: qué son, de dónde vienen y cómo pueden ser comparados con otros procedimientos de búsqueda.

3.1. ¿Qué son los algoritmos genéticos?

Los algoritmos genéticos [10] son algoritmos de búsqueda basados en los mecanismos de la selección natural y la genética. Combinan métodos en los que intervienen la supervivencia de los individuos más aptos de la población con otros que son puramente aleatorios. En cada generación, un nuevo conjunto de individuos es creado usando características de los mejores individuos de la generación anterior.

Los algoritmos genéticos han sido desarrollados por John Holland (1975) y sus compañeros de trabajo en la Universidad de Michigan. Los objetivos de su búsqueda han sido dos:

1. Abstraer y explicar rigurosamente el proceso adaptativo de los sistemas naturales.

2. Diseñar sistemas software artificiales basados en los mecanismos de los sistemas naturales.

Este enfoque ha permitido importantes descubrimientos en ambas áreas, tanto en los sistemas científicos naturales como en los artificiales.

En la naturaleza los individuos de una población compiten entre sí para buscar cosas materiales como puedan ser alimento, agua, un lugar para refugiarse... así como pueden competir en la búsqueda de un compañero para poder reproducirse. Aquellos individuos con mayor éxito en sobrevivir y atraer compañeros tienen mayor probabilidad de generar un mayor número de descendientes. Por el contrario, individuos poco dotados producirán un menor número de descendientes. De esta forma, los genes de los mejores individuos se propagarán en sucesivas generaciones mientras que los genes de los peor adaptados se irán perdiendo paulatinamente. La combinación de buenas características de generaciones anteriores puede llegar a producir incluso "superindividuos", cuya adaptación es mucho mayor que los producidos anteriormente. Así, las especies evolucionan consiguiendo individuos cuyas características están cada vez más adaptadas al medio en el que viven.

Los Algoritmos Genéticos usan una analogía directa con el comportamiento natural. Trabajan con una población de individuos, cada uno de los cuales representa una solución factible a un problema dado. A cada individuo se le asigna un valor o aptitud, llamado fitness, que nos dice cuánto de bueno es un individuo. Cuanto mayor sea este valor, mayor será el grado de adaptación del individuo y la probabilidad de que dicho individuo sea seleccionado para reproducirse con otro será mayor también. Por tanto, este posible cruce producirá nuevos individuos, descendientes de los anteriores, que estarán más adaptados al medio. De la misma forma, cuanto menor sea el valor del fitness, menor será el grado de adaptación del individuo y la probabilidad de que el individuo sea seleccionado para reproducirse será menor también. Como dijimos anteriormente, los individuos menos adaptados tienden a perderse con el paso del tiempo.

Si el Algoritmo Genético ha sido bien diseñado, la población convergerá hacia una solución óptima del problema, ya que las mejores características de los individuos se han ido propagando a través de la población. Tendiendo la generación siguiente a ser mejor que la anterior.

El poder de los Algoritmos Genéticos proviene del hecho de que se trata de una técnica robusta y pueden tratar con éxito una gran variedad de problemas. A pesar de que no se puede asegurar que lleguen a encontrar la solución óptima, existe evidencia empírica de que se encuentran soluciones de un nivel aceptable, en un tiempo competivo con respecto al resto de algoritmos de combinación aleatoria.

En los casos en los que existan técnicas especializadas para resolver un determinado problema, lo más probable es que superen al Algoritmo Genético, tanto en rapidez como en eficacia. El campo de aplicación de los Algoritmos Genéticos es más general, usándose para resolver aquellos problemas en los que no exista una técnica especializada. Pero incluso se puede mezclar la técnica específica, en el caso que exista, con técnicas propias de Algoritmos Genéticos, consiguiendo obtener mejoras en los resultados.

Los Algoritmos Genéticos son diferentes respecto al resto de procedimientos de búsqueda en:

1. Los Algoritmos Genéticos trabajan con un conjunto de parámetros codificados, no con los parámetros concretos.

2. Los Algoritmos Genéticos utilizan un subconjunto del espacio total para obtener información sobre el universo de búsqueda, a través de las evaluaciones de la función a optimizar.

3. Los Algoritmos Genéticos usan información extraída de una función objetivo, no derivada de otro tipo de conocimiento.

4. Los Algoritmos Genéticos usan reglas de transición probabilística, no reglas deterministas.

3.2. Elementos y operadores de un Algoritmo Genético

Ahora pasaremos a explicar algunos elementos básicos que entran en juego a la hora de diseñar un algoritmo genético.

3.2.1. Población inicial

Cuando hablamos de Algoritmos Genéticos, a menudo nos referimos a diversas generaciones o poblaciones de indiviudos presentes durante la ejecución del algoritmo. Se ha hablado hasta ahora de que los mejores individuos de una población tendrán más posiblidades de reproducirse con otros. Así, se obtendrá una nueva generación en la que los individuos que la componen estarán mayor adaptados al medio que los de la generación anterior. Pero, ¿y la primera generación?

La primera generación con la que se trabaja, o población inicial, estará formada por aquellos individuos generados aleatoriamente, representado una posible solución al problema. En el caso de no obtener dicha población de manera aleatoria, habrá que garantizar, al menos, que los individuos que la componen forman un subconjunto diverso, existiendo representantes de la mayor parte de la población posible en el subcojunto elegido. De esa manera evitaremos una convergencia demasiado rápida.

3.2.2. Función objetivo

Un aspecto muy importante en el comportamiento de los Algoritmos Genéticos es la adecuada elección de la función objetivo. Idealmente nos interesaría construir funciones objetivos que verifiquen por ejemplo que para dos individuos que se encuentren cercanos en el espacio de búsqueda, sus valores obtenidos por la función objetivo sean similares. Podemos encontrarnos también con que el Algoritmo Genético posea una gran cantidad de óptimos locales o que el valor óptimo global se encuentre muy aislado.

La regla general para construir una buena función objetivo es que debe reflejar el valor del individuo de una manera real. Pero en muchos problemas de optimización, con una gran cantidad de restricciones, buena parte de los individuos de búsqueda representan individuos no válidos.

Para solventar estos problemas que surgen con la existencia de individuos sometidos a restricciones se han propuestos varias soluciones. Por ejemplo, la absolutista, en la que los individuos que no verifican las restricciones, no son considerados como tales, y se siguen efectuando cruces y mutaciones hasta obtener individuos válidos o se les asigna

a estos individuos un valor de 0.

Otra opción consiste en reconstruir los individuos que no verifican las restricciones. Se crea un nuevo operador llamado reparador que se encarga de reconstruir el individuo.

Otro enfoque está basado en la penalización de la función objetivo. La idea general consiste en dividir la función objetivo del individuo entre una cantidad (la penalización) que guarda relación con las restricciones que dicho individuo viola. Dicha cantidad puede simplemente tener en cuenta el número de restricciones violadas o bien el denominado costo esperado de reconstrucción, el coste asociado a la conversión de dicho individuo en otro que no viole ninguna restricción.

Otra técnca es la denominada evaluación aproximada de la función objetivo. Para algunos casos, resulta mejor la obtención de n funciones objetivo aproximadas que la evaluación exacta de una única función objetivo.

3.2.3. Operadores de selección

Los operadores de selección serán los encargados de escoger los individuos que tomen parte en la reproducción y los que no. Primero deberemos calcular aplicando la función objetivo la aptitud o fitness de cada inidividuo y una vez realizado este paso podremos aplicar los diferentes operadores que hayamos definido de selección.

Existen varios mecanismos de selección, los cuales pasaremos a explicar a continuación.

- **Selección por ruleta**: también llamado selección de Montecarlo. Fue propuesto por DeJong y es posiblemente el método más utilizado desde los orígenes de los Algoritmos Genéticos.

  Este método permite que los mejores individuos tengan mayor probabilidad para ser elegidos, pero al mismo tiempo, que los peores individuos puedan ser también elegidos. De esta forma mantenemos la diversidad en la población. A cada uno de los individuos de la población se le asigna una parte proporcional al tamaño de una ruleta, en función de la probabilidad de ser escogido dicho individuo, de

forma que la suma de todos los porcentajes sea la unidad. Como se puede deducir, los mejores individuos recibirán una porción mayor de la ruleta que los peores.

Ordenamos la población en base a la probabilidad de que cada individuo sea elegido por lo que nos encontraremos los mejores individuos al principio de la ruleta y los peores al final. Para seleccionar un individuo basta con generar un número aleatorio en el intervalo [0..1] y devolver el individuo situado en dicha posición de la ruleta.

Teniendo en cuenta los valores presentes en el ejemplo de la figura 3.1. Si el valor generado por el número aleatorio es por ejemplo el 0.4, el individuo seleccionado sería el individuo 2 en este caso.

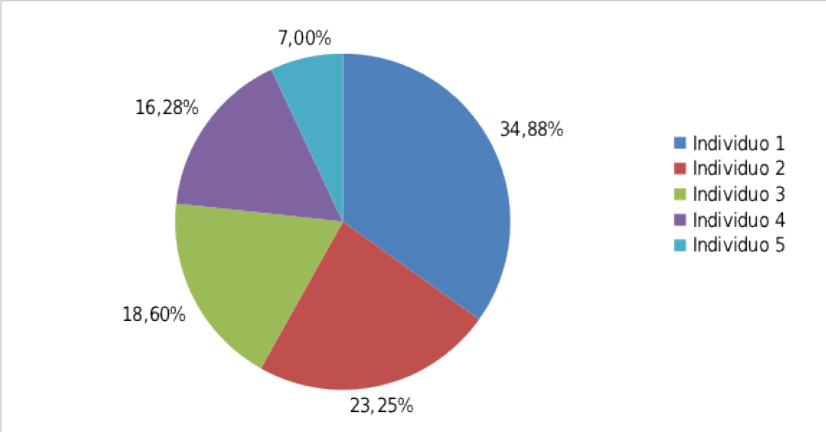

Figura 3.1.: Selección por ruleta

- **Selección por torneo**: la idea principal de este método consiste en realizar un torneo o comparación entre un pequeño subcojunto de individuos llamado el tamaño del torneo. Existen dos versiones de selección mediante torneo:
  - Determinística.
  - Probabilística.

  En la versión determinística se selecciona al azar un número n de individuos. De entre los n individuos seleccionados se elige al más apto, el cual se pasa a la si-

guiente generación.

En la versión probabilística en vez de elegir siempre al mejor individuo, se genera un número aleatorio entre [0..1] y si el valor generado es mayor que un parámetro p fijado se escoge al mayor, y en caso contrario a alguno de los otros.

Variando el número de participantes en cada torneo se puede modificar la presión de selección. Cuando participan muchos individuos en un torneo, la presión es elevada y la probabilidad de que un individuo poco apto sea escogido es prácticamente nula.

- **Selección por truncamiento**: en esta selección las soluciones candidatas son ordenadas según su función objetivo y una proporción p de los individuos con mejor aptitud es seleccionada y reproducida 1/p veces. Esta selección es menos sofisticada que la mayoría de los métodos de selección, y generalmente no es usada en la práctica.

- **Selección basada en ranking**: En esta selección los individuos se ordenan según su aptitud y luego son asignados con una segunda medida, inversamente proporcional a su posición en el ranking (otorgando una mayor probabilidad a los mejores). Los valores de esta segunda asignación pueden ser lineales o exponenciales. Finalmente, los individuos son seleccionados proporcionalmente a esta probabilidad.

  Este método disminuye el riesgo de convergencia prematura que se produce cuando se utiliza selección de ruleta en poblaciones con unos pocos individuos con unos valores de aptitud muy superiores al resto.

- **Selección aleatoria**: para mantener un porcentaje de aleatoreidad en la selección de individuos como puede ocurrir en la realidad con cualquier especie, se estable esta selección. Con la selección aleatoria nos olvidamos de si un individuo tiene mejor valor de aptitud que otro, es decir, consideramos todos los individuos iguales, con la misma probabilidad de ser elegidos y escogemos uno de ellos.

3.2.4. Operadores genéticos

Una vez elegidos los individuos de la población, pasaremos a explicar el siguiente paso, el intercambio de material genético entre ellos para dar lugar a nuevos individuos, posiblemente mejor adaptados al medio. Las operaciones de reproducción que se explicarán son el cruce y la mutación.

Operador de cruce

El objetivo principal que se obtiene tras aplicar este operador es obtener dos individuos nuevos, llamados *hijos*, una vez se ha combinado la información genética de sus *padres* o *progenitores*. Los hijos heredan las características de los padres, por tanto, la descendencia, o al menos parte de ella, debería tener una mejor aptitud que sus progenitores después de que estos compartan sus características buenas.

Este operador no se aplica a todos los pares de individuos que componen la población seleccionada. Existe habitualmente, la llamada *probabilidad de cruce* que teniendo un valor entre [0..1] nos indica la probabilidad de aplicar dicho operador a cada pareja de individuos. Si no se aplica el operador, los hijos se obtienen duplicando la información genética de los padres, resultando ser los hijos idénticos a los padres.

Existen diversos métodos con los que aplicar el cruce:

- **Cruce de un punto:** Se genera un número aleatorio que determinará la posición a partir de la cual realizaremos el cruce de los individuos. De esta forma, cada uno de los hijos heredará información de ambos partes, produciéndose la recombinación. En la figura 3.2 podemos ver un ejemplo de cruce producido a partir de la posición 5.

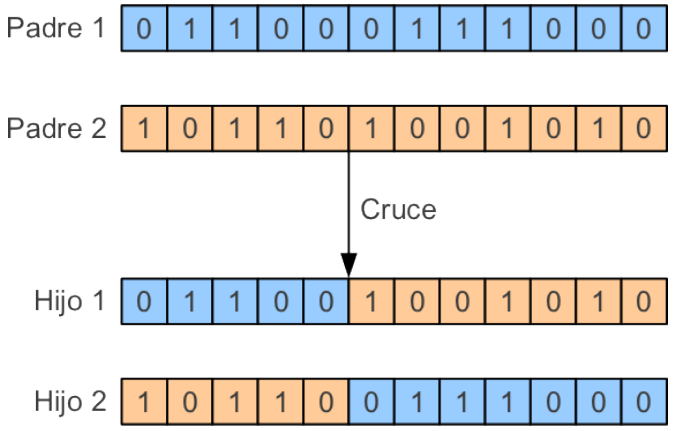

Figura 3.2.: Cruce de un punto

- **Cruce multipunto:** Este tipo de cruce cruzaremos a los individuos a partir de 2 o más puntos, en este caso lo explicaremos para 2. Se generan dos números aleatorios, no pudiendo coincidir estos con los extremos del individuo. Realizaremos el cruce entre dos posiciones delimitadas por los dos números aleatorios generados. En la figura 3.3 podemos ver un ejemplo de cruce producido entre las posiciones 3 y 9.

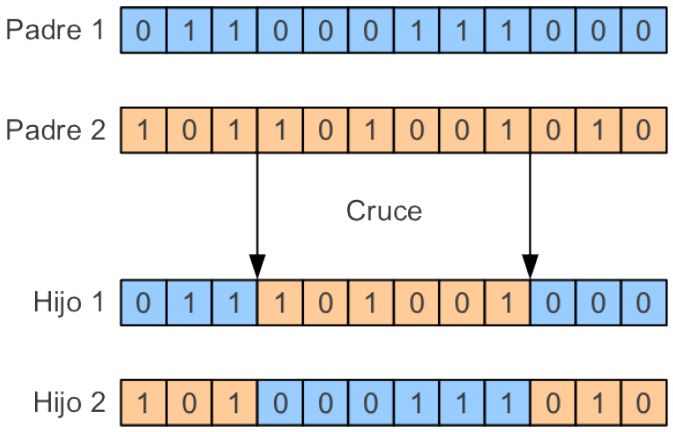

Figura 3.3.: Cruce multipunto

- **Cruce uniforme:** Aplicamos una máscara binaria que nos indicará qué posición de

cada progenitor pasará a uno u otro hijo. Para cada posición tendremos un valor de máscara, si este valor es un 1, el valor en dicha posición del hijo 1 se corresponderá con la información del padre 1, mientras que si el valor de máscara se corresponde con un 0, entonces el hijo 1 obtendrá el valor correspondiente al del padre 2. El hijo 2 se obtendrá intercambiando los progenitores, justo de la forma contraria a como se ha obtenido el hijo 1.

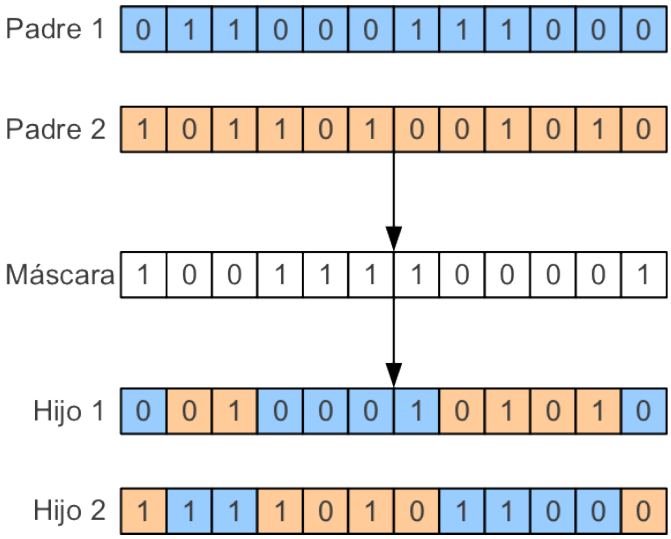

Figura 3.4.: Cruce uniforme

Operador de mutación

El objetivo principal de este operador consiste en alterar alguno de los componentes de un individuo. Se aplica, al contrario que el operador de cruce de manera individual sobre cada uno de los hijos.

Se elige uno de los componentes del individuo a mutar aleatoriamente y se genera un nuevo valor válido para el componente del individuo seleccionado. Podemos generar ese nuevo valor aleatoriamente entre alguno de los valores permitidos o bien aplicando alguna relación entre el valor que tenía con anterioridad, una *constante de mutación* definida como parámetro de entrada y la *probabilidad de mutación*.

En la figura 3.5 podemos ver un ejemplo de mutación producida en la posición 6 del individuo.

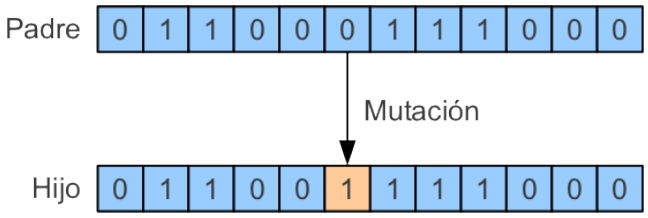

Figura 3.5.: Operador de mutación

3.2.5. Condiciones de terminación

El Algoritmo Genético se debería detener idealmente cuando se alcanzase la solución óptima al problema, pero ésta generalmente se desconoce, por lo que se deben utilizar otros criterios de parada que garanticen que se ha ejecutado un número de veces lo suficientemente alto, unos resultados que pueden calificarse de buenos o que ya no están mejorando los individuos generados.

Algunos de esos criterios de parada son:

Número de generaciones máximo

Consiste en realizar de manera repetida el mismo proceso: cálculo del fitness de la generación anterior, seleccionar individuos, generar nuevos individuos, aplicar los operadores genéticos... así hasta llegar al número de generaciones establecidas como parámetro de entrada. Con este criterio no nos fijamos en los resultados que vayamos obteniendo, sólo en las veces que tenemos que repetir el mismo proceso.

Al menos el valor deberá ser 1, ya que siendo 0 no se generaría ninguna población.

Convergencia del Algoritmo Genético

Útil para los casos en los que podamos determinar cómo de buena es la población que se acaba de generar. Se puede configurar un porcentaje por defecto y si la población

satisface al menos en ese porcentaje lo buena que sería la solución óptima podemos parar la ejecución. De esta forma aseguramos llegar a generar una población que cumple unos requisitos mínimos sobre la solución que consideramos la que sería ideal.

En nuestro caso, el porcentaje de mutantes muertos sería el factor que determina cuánto de buena es la población generada. Si alcanzamos o superamos el porcentaje de mutantes muertos establecido, detenemos la ejecución.

Estancamiento en la evolución del fitness máximo

Consiste en ver si el fitness del mejor individuo de la población va evolucionando. Lo ideal sería que este valor aumentase tras cada generación y consigamos en cada iteración, al menos un individuo mejor que el mejor conseguido en la iteración anterior.

Configuramos un parámetro, que será el número de generaciones consecutivas que puede estar el algoritmo sin mejorar el fitness. Si mientras ejecutamos el algoritmo nos llevamos ese número de generaciones sin mejorar el fitness del mejor individuo paramos la ejecución.

Cabe mencionar que en el momento en el que en una generación se mejore el fitness del mejor individuo generado, la cuenta se pondrá a 0, pues hablamos de generaciones consecutivas sin mejorar el fitness.

Estancamiento en la evolución del fitness medio

Es igual que el criterio anterior pero en lo que nos fijamos en cada iteración no será en el fitness del mejor individuo, sino en el fitness medio de la población. Si durante un número prefijado de veces, el fitness medio de la población no mejora, entenderemos que a la solución, a priori óptima del problema, no podemos llegar.

AL igual que en el caso anterior, también hablamos de generaciones consecutivas.

3.2.6. Esquema de un Algoritmo Genético

En el siguiente ejemplo podemos observar cómo funciona un algoritmo básico.

Partimos de un tamaño de población *tam* y un número máximo de generaciones *fin* que se corresponderá con el criterio de parada. En primer lugar, se crea una población inicial de individuos aleatorios y se calcula el fitness. Mientras no hayamos alcanzado el número de generaciones fijado al inicio, la población irá sufriendo numeros cambios en sus individios. Seleccionaremos 2 invididuos en cada ocasión de la población inmediatamente anterior, a los que les aplicaremos los operadores de cruce y mutación. Estos nuevos individuos pasarán a formar parte de la nueva generación. Cuando hayamos alcanzado en esta nueva generación el tamaño dado como parámetro de entrada, calcularemos el fitness de todos ellos y repetiremos el proceso tantas veces como número de generaciones hayamos indicado que hay que generar.

Algoritmo 1 Esquema de un Algoritmo Genético

Entrada: $tam > 0 \wedge fin > 0$

Salida: Población solución

$nueva \leftarrow generar\_poblacion\_incial(tam)$

$calcular\_fitness(nueva)$

$numGen \leftarrow numGen + 1$

mientras $numGen < fin$ **hacer**

$numGen \leftarrow numGen + 1$

$cont \leftarrow 0$

$P \leftarrow nueva$

$nueva \leftarrow 0$

mientras $cont < tam$ **hacer**

$select \leftarrow seleccionar\_dos\_individuos(P)$

$aplicar\_cruce(select)$

$aplicar\_mutacion(select)$

$insertar\_en\_nueva\_poblacion(select, nueva)$

$cont \leftarrow cont + 2$

fin mientras

$calcular\_fitness(nueva)$

fin mientras

devolver $nueva$

CAPÍTULO 4

ANÁLISIS

En este capítulo realizaremos el análisis de la aplicación; las necesidades que debe cumplir el proyecto, los requisitos del sistema a desarrollar. A partir de ellos se definirá el diagrama de casos de uso y el modelo conceptual.

4.1. Requisitos funcionales

Los requisitos funcionales nos indican las necesidades que debe cumplir el sistema.

Si nos remontamos a aplicaciones anteriores, en *ServiceAnalyzer* se creó un analizador de servicios Web, que generaba plantillas parametrizadas para producir mensajes de acuerdo a todas las restricciones impuestas desde WSDL, XML Schema y y el WS-I Basic Profile 1.1.

En *TestGenerator* se crea un generador aleatorio de los datos que necesitaban esas plantillas. Además, se crea un lenguaje de dominio específico, capaz de representar el conjunto de datos y restricciones impuestas por *ServiceAnalyzer*.

Partiendo de ahí, para generar un conjunto parametrizado de casos de prueba de *BPELUnit*, como es el objetivo de *GAmeraHOM-ggen*, usaremos el generador aleatorio de datos de *TestGenerator* y el lenguaje de dominio específico (*spec*) que en él se creó.

El generador implementado en *TestGenerator* nos servirá para generar de forma aleatoria los datos de los casos de prueba de la primera población. Usaremos el lenguaje *spec* para definir en un fichero externo los diferentes tipos y restricciones de las variables de la composición a estudiar. A partir de ahí y aplicando un algoritmo genético se generarán los casos de prueba más robustos y resistentes posibles.

En el diagrama 4.1 podemos ver de manera general los pasos que se siguen hasta conseguir el nuevo conjunto de casos de prueba. *ServiceAnalyzer* recibe una composición BPEL y genera unas plantillas que deben ser rellenadas. El fichero con extensión *spec* nos servirá para definir el conjunto de datos y restricciones contemplados en *ServiceAnalyzer*. *GAmeraHOM-ggen*, con ayuda de *TestGenerator* generará la primera población de casos de prueba aleatorios. Posteriormente, tras aplicársele a esta primera población el algoritmo genético correspondiente implementado en *GAmeraHOM-ggen* durante el número de generaciones oportuno en cada caso, conseguiremos generar nuevos casos de prueba, más robustos y mejor adaptados.

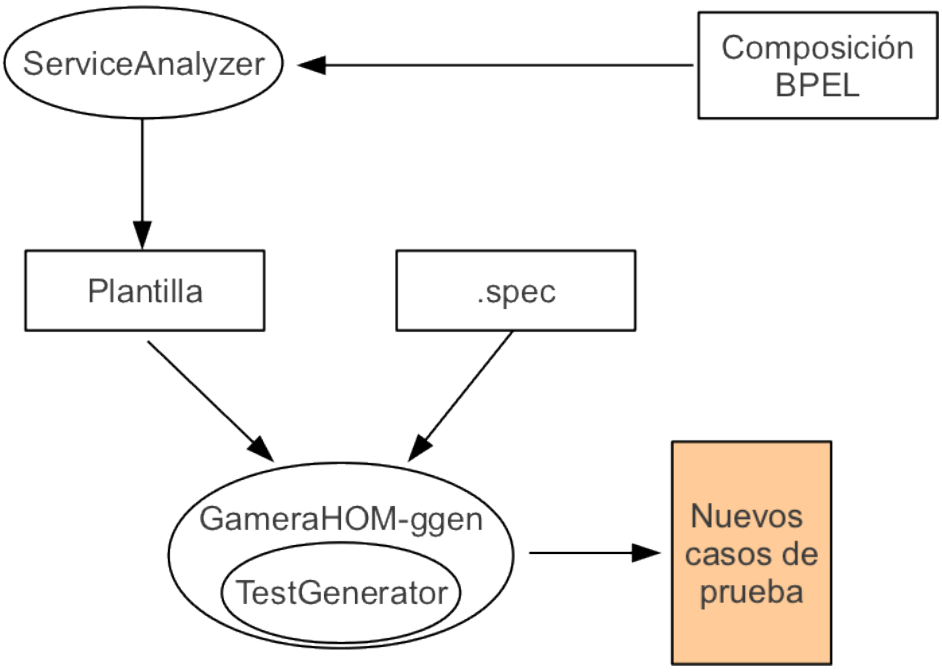

Figura 4.1.: Componentes usados

El sistema deberá ser capaz de exportar los diferentes individuos que se hayan ido generando durante la ejecución del algoritmo, en formatos compatibles con *BPELUnit*. A pesar de que son compatibles tanto el formato CSV como Apache Velocity, se opta tan sólo por presentar los datos en formato Apache Velocity por ser éste más potente que CSV. Aunque CSV es muy sencillo y fácil de usar, hay tipos como por ejemplo la lista de elementos que no puede representar.

En Apache Velocity las variables y sus valores serán representados de la forma que sigue: `#set($NombreVariable = [valor1, valor2])` donde `valor1`, `valor2`... representará el valor de la variable para el caso de prueba 1, 2... respectivamente. Como puede intuirse para cada variable se generará una línea que representará los diferentes valores que tome dicha variable para cada caso de prueba. En la sección 1.4.5 tenemos un ejemplo de una posible salida del programa en formato Apache Velocity.

Teniendo esto presente, se definen los siguientes requisitos funcionales.

- Generar los tipos y restricciones de las variables de los casos de prueba.
- Generar una población inicial de casos de prueba aleatorios.
- Analizar cuánto de buenos son los casos de prueba generados, calcular el fitness.
- Crear generadores de nuevos individuos.
- Aplicar operadores de selección para escoger individuos de una población.
- Aplicar operadores genéticos para crear nuevos individuos, en teoría, mejor adaptados al medio.
- Comprobar si se cumplen las condiciones para detener el algoritmo.

4.2. Requisitos de implementación

Los requisitos de implementación impuestos por el grupo UCASE son los siguientes:

- El lenguaje a utilizar para la implementación del proyecto deberá ser Java [11]. Esto es debido a que la mayoría de las aplicaciones del grupo, actualmente están escritas en dicho lenguaje, por lo que de esta forma se facilita la reutilización de código..

- Se exige la realización de pruebas paramétricas para poder evaluar si el funcionamiento de cada uno de los componentes funciona como se espera. El framework exigido es JUnit [13].

- El grupo de investigación UCASE exige también para la realización de todos sus proyectos, utilizar un entorno de integración continua. Gracias a ello se podrá llevar un mejor control de los distintos trabajos que se estén realizando y poder verificar así de manera periódica que todo funciona como se espera.

- Se exige también la posibilidad de poder configurar los parámetros de entrada de la aplicación mediante un fichero de texto externo escrito en un lenguaje concreto, YAML [12].

4.3. Atributos del sistema

Las propiedades que el sistema software deberá cumplir son:

- Mantenibilidad: es fundamental para que posibles correcciones se hagan rápidas y en un futuro que este trabajo pueda ser ampliado.

- Fiabilidad: Los resultados de las pruebas de mutaciones que se den no pueden contener ningún tipo de error, puesto que será fundamental para poder realizar el análisis de las mutaciones realizadas en fututos estudios estadísticos.

- Facilidad de uso: a pesar de que el proyecto está dirigido a personas con conocimientos informáticos elevados, se intentará facilitar su uso en la medida de lo posible para que no se haga demasiado tedioso trabajar con la aplicación.

- Licencia libre: para que la aplicación tenga el mejor uso público posible se le otorga dicha licencia.

4.4. Casos de uso

Los casos de uso describen los pasos o actividades que deberán realizarse para llevar a cabo un determinado proceso. Los personajes o entidades que participarán en un caso de uso se denominan actores. En ingeniería del software, se define un caso de uso como una secuencia de interacciones que se desarrollarán entre un sistema y sus actores en respuesta a un evento que inicia un actor principal sobre el propio sistema.

Diagrama de casos de uso

El diagrama que sirve para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios y/o con otros sistemas se conoce como diagramas de casos de uso. El diagrama de *GAmeraHOM-ggen* podemos verlo en la figura 4.2.

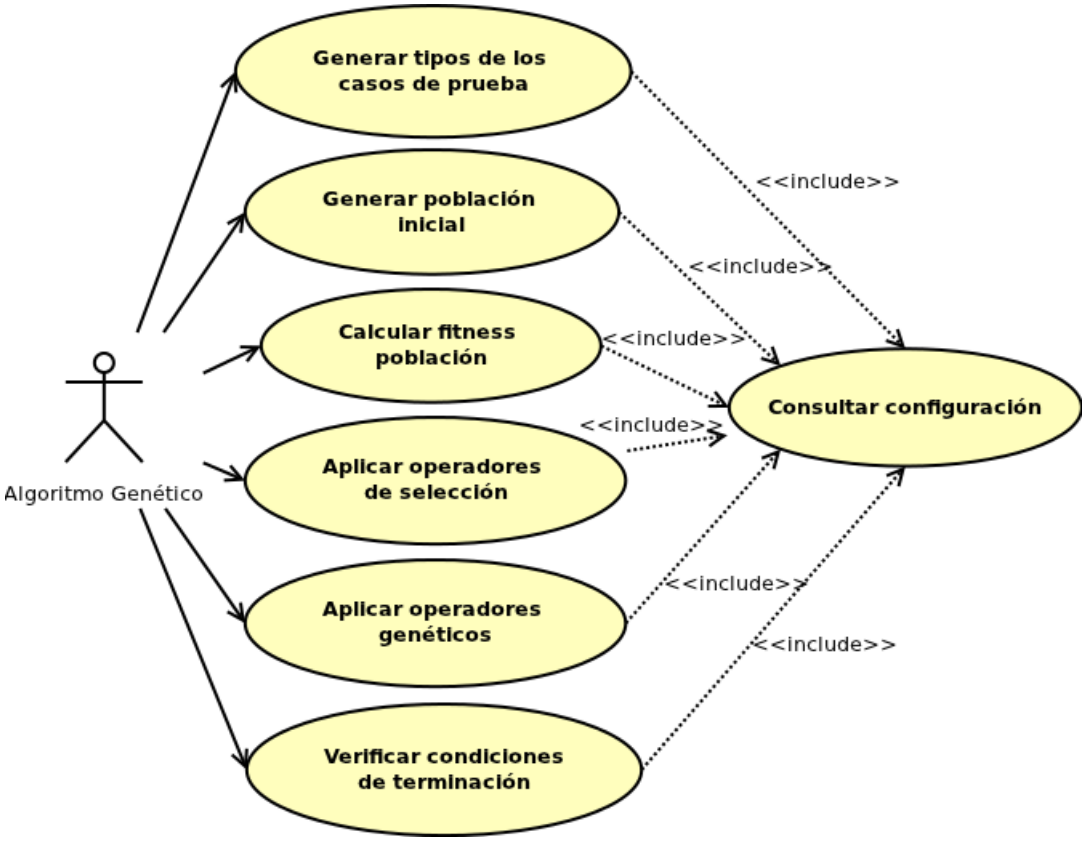

Figura 4.2.: Diagrama de casos de uso

A continuación se especifica el funcionamiento de cada uno de los casos de uso representados en la figura 4.2.

Generar tipos casos de prueba

Actor principal Algoritmo genético

Precondiciones El fichero con la declaración de los tipos y de las restricciones debe existir y no estar vacío.

Postcondiciones Devuelve una lista con los distintos tipos de las variables que intervienen en la composición WS-BPEL.

Escenario principal

1. El algoritmo genético solicita la creación de los diferentes tipos que intervienen en una composición WS-BPEL.
2. Se consulta el fichero externo en donde están declarados los tipos con sus restricciones correspondientes.
3. El sistema devuelve la lista de los tipos a usar incluyendo las restricciones que poseen cada uno de ellos.

Generar población inicial

Actor principal Algoritmo genético

Precondiciones El fichero de configuración está cargado y la lista con los tipos de las variables que componen los casos de prueba está creada, así como el generador de datos a usar.

Postcondiciones Devuelve una población inicial aleatoria.

Escenario principal

1. El algoritmo genético solicita la creación de una población inicial.
2. Se consulta el tamaño total de la población del fichero de configuración.
3. Se generan individuos de manera aleatoria según la lista de tipos creada hasta completar la población total.

Variaciones

2a. El tamaño de la población es menor o igual a 0.

1. El sistema informa que el tamaño de la población no puede ser menor o igual a 0 y cancela el caso de uso.

Calcular fitness población

Actor principal Algoritmo genético

Precondiciones Una población ya se ha creado y el fichero de configuración está cargado.

Postcondiciones Devuelve el fitness de cada individuo de la población.

Escenario principal

1. El algoritmo genético solicita el cálculo del fitness de los individuos que componen la población.
2. Se consultan las características necesarias del fichero de configuración.
3. Se generan los diferentes mutantes dada una composición BPEL.
4. Se ejecutan los casos de prueba generados contra todos los mutantes y se comparan las salidas generándose la llamada matriz de ejecución.

   En dicha matriz de ejecución se indicará para cada caso de prueba y cada mutante la diferencia existente entre ellos marcando cada posición con un 0 si no existen diferencias (mutante vivo), un 1 si existen (mutante muerto) o un 2 si el mutante es erróneo.
5. El fitness de los individuos es calculado a partir de la matriz de ejecución.

Aplicar operadores de selección

Actor principal Algoritmo genético

Precondiciones Debe existir al menos un operador de selección definido en el fichero de configuración y una población creada.

Postcondiciones Devuelve los individuos de la población seleccionados.

Escenario principal

1. El algoritmo genético solicita el conjunto de individuos de la población que serán seleccionados por el operador o los operadores de selección definidos.
2. Se consultan las características necesarias del fichero de configuración.

3. Para el porcentaje de individuos indicado por cada operador de selección, el sistema devuelve la población seleccionada, ya sea de manera aleatoria, tras aplicar el método de la ruleta, etc.

Aplicar operadores genéticos

Actor principal Algoritmo genético

Precondiciones Debe existir al menos un operador genético definido en el fichero de configuración y una población de individuos sobre los que aplicar los operadores.

Postcondiciones Nuevos descendientes de la población anterior.

Escenario principal

1. El algoritmo genético solicita aplicar los operadores genéticos definidos en el sistema.
2. Se consultan las características necesarias del fichero de configuración.
3. El algoritmo genético selecciona individuos de la población de dos en dos mientras siga habiendo alguno y aplica los operadores genéticos definidos en el sistema sobre ellos.
4. El sistema devuelve la población formada por descendientes de la inmediatamente anterior.

Verificar condiciones de terminación

Actor principal Algoritmo genético

Precondiciones Debe existir al menos una condición de terminación.

Postcondiciones Devuelve si debemos detener la ejecución o no del algoritmo.

Escenario principal

1. El algoritmo genético solicita comprobar si se dan las condiciones o no para detener la ejecución del algoritmo.
2. Se consultan las características necesarias del fichero de configuración.
3. El algoritmo genético comprueba para cada condición de terminación si ésta se cumple, en cuyo caso detenemos la ejecución del algoritmo.

Consultar configuración

Actor principal Algoritmo genético

Precondiciones El fichero de configuración existe en el sistema.

Postcondiciones Ninguna.

Escenario principal

1. El sistema devuelve al algoritmo genético todas las características del fichero de configuración.

4.5. Modelo conceptual de datos

El modelo conceptual de datos describe las estructuras de datos de un sistema y sus restricciones de integridad. Está orientado a representar los elementos que intervienen en un problema concreto y las relaciones existentes entre ellos.

En la figura 4.3 se muestra el modelo conceptual de datos de *GAmerHOM-ggen*, el cual se ha realizado siguiendo la notación de UML 2.0 [14].

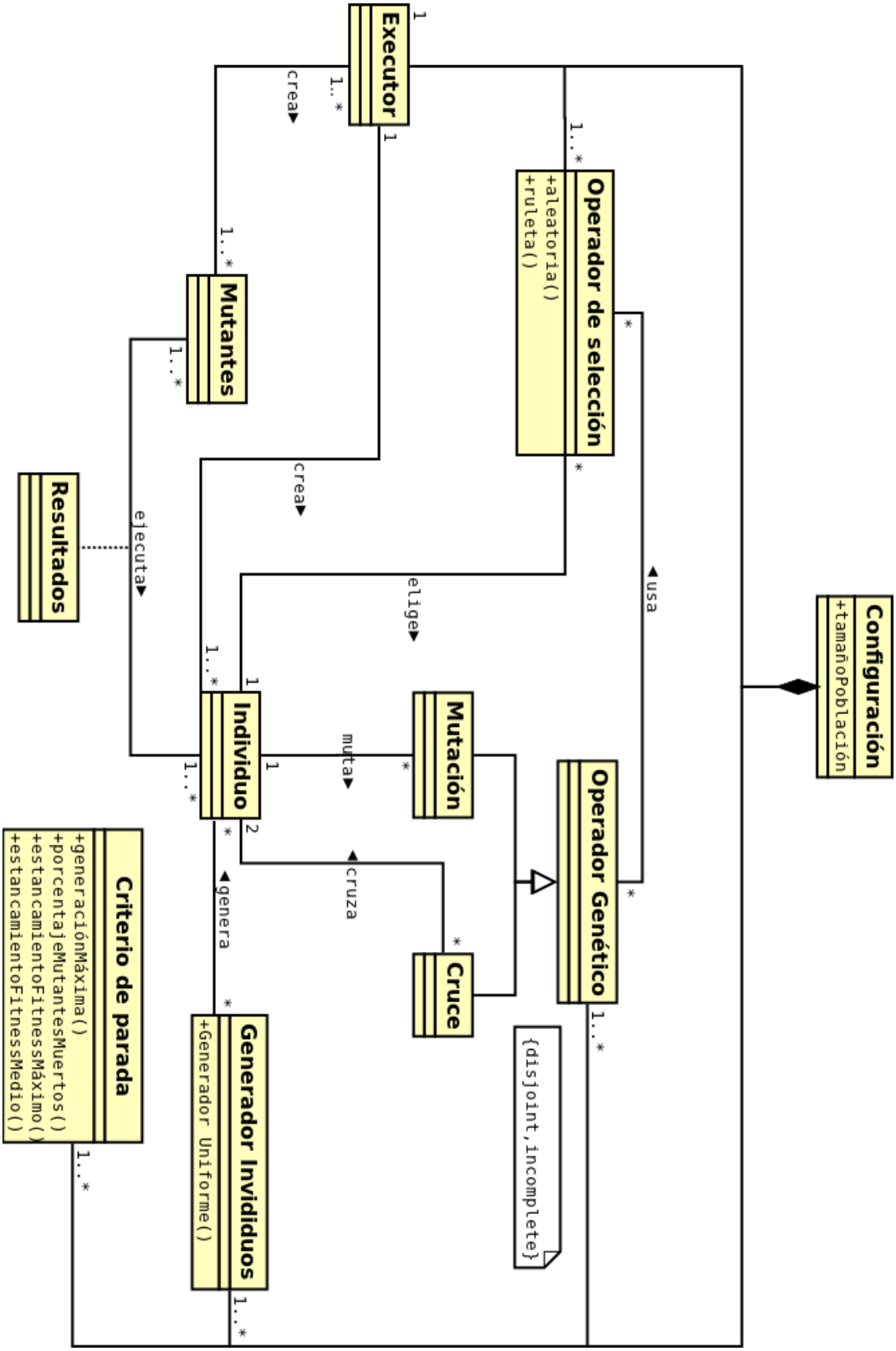

Figura 4.3.: Modelo conceptual de datos

- **Configuración** agrupará toda la parametrización de *GAmeraHOM-ggen* conteniendo atributos tales como el tamaño de la población, los operadores de selección de individuos, los operadores genéticos implementados, mutación y cruce, los generadores de nuevos individuos, las condiciones de terminación del algoritmo y el ejecutor con el que trabajaremos.

- **Operador de selección** contendrá los diversos métodos disponibles para dada una población seleccionar un conjunto de individuos.

- **Operador genético** proporcionará los diversos operadores de reproducción existentes. Mediante el cruce y la mutación obtendremos individuos más aptos en cada ocasión.

- **Executor** representa el programa con el cual queramos trabajar. En nuestro caso, que trabajaremos con *BPELExecutor*, definiremos la composición original, la plantilla en donde se definen los casos de prueba y un fichero de salida. Entrará en juego en un primer momento para analizar la composición original y crear los diferentes mutantes a partir de este análisis. Posteriormente, con ayuda de toda esta información generada, generaremos nuestros individuos, los casos de prueba concretos.

- **Mutante** representa el programa que ha sufrido los diferentes cambios con respecto al original (en este caso uno sólo, ya que trabajaremos con mutantes de orden 1). Sobre los mutantes cargaremos también los diferentes casos de prueba.

  Compararemos las salidas producidas tras ejecutar los casos de prueba contra todos los mutantes y el programa original almacenando los resultados en *Resultados*.

- **Individuo** será el centro de la herramienta. Su estructura vendrá determinada por los tipos con los que trabaje cada composición. Guardaremos los tipos y el valor de cada uno de ellos formando todos juntos el individuo.

  Un conjunto de individuos conforman una población, su tamaño vendrá determinado por un parámetro configurado inicialmente.

CAPÍTULO 5

DISEÑO

Pasaremos a explicar en este capítulo algunas condiciones que nos llevarán a tomar ciertas decisiones de diseño en el proyecto y la estructura que el mismo seguirá. Además de ello, explicaremos cómo se integra la herramienta dentro de los componentes ya existentes en el grupo.

5.1. Arquitectura de GAmera

En capítulos anteriores hemos explicado que se hace necesario analizar el programa original para ver las instrucciones del mismo que se pueden cambiar para dar lugar a los diferentes mutantes. Una vez que los mutantes se han generado los ejecutaremos contra los casos de prueba generados por nuestra herramienta *GAmeraHOM-ggen*. A continuación explicaremos cómo se realiza este proceso.

GAmera [15] es la herramienta para la generación y ejecución autómatica de mutantes para composiciones de Servicios Web en WS-BPEL [16]. Está constituida por tres componentes principales como podemos observar en la figura 5.1.

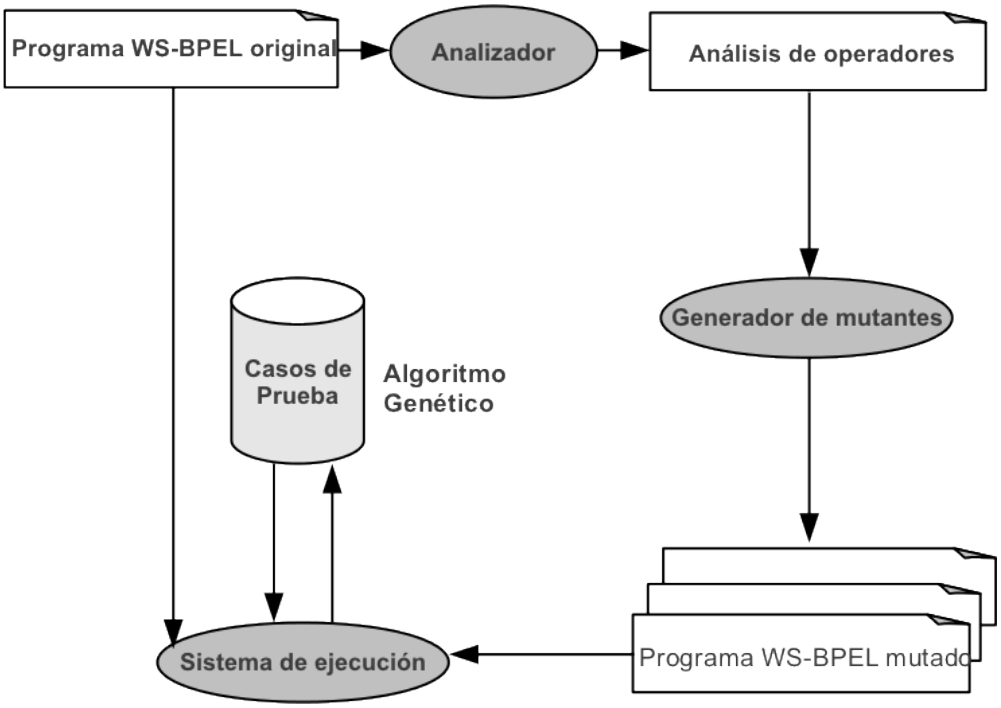

Figura 5.1.: Arquitectura *GAmera*

- **Analizador**: es el encargado, como su propio nombre indica, de la fase de análsis y es el componente que actúa primero. Recibe como entrada la composición WS-BPEL a probar y determina los operadores de mutación que se pueden aplicar, es decir, las instrucciones que pueden ser modificadas sobre dicha composición y cómo pueden ser modificadas. Por ejemplo, el operador de mutación encargado de cambiar un operador lógico por otro, como explicamos en 1.4.6.

  Aunque *GAmera* puede trabajar con mutantes de orden superior a 1, es decir, un mismo mutante posee más de una diferencia con respecto al programa original. En *GAmeraHOM-ggen* se impone la restricción de trabajar únicamente con mutantes de orden 1, pudiéndose ampliar en un futuro el poder trabajar con mutantes de órdenes superiores.

- **Generador de mutantes**: es el siguiente componente que entra en acción. Partiendo de la información proporcionada por el analizador, se crea la estructura

de los cambios a aplicar en cada mutante codificada de la siguiente forma (ver figura 5.2). En un primer campo llamado operador, se codifica al operador con un valor entero (en un rango que va entre 1 y el número máximo de operadores existentes); el campo instrucción nos dará información acerca del número de instrucción sobre la que aplicaremos el operador de mutación; y por último, el campo atributo nos informará del nuevo valor que tomará el elemento de la instrucción al ser modificado por el operador de mutación.

Figura 5.2.: Estructura de un mutante

- **Sistema de ejecución**: es el último componente a utilizar. Va generando los mutantes con la información que recibe de la fase anterior y los ejecuta contra un conjunto de casos de prueba, resultando como posible salida que el mutante esté vivo, muerto o sea erróneo, como hemos comentando ya en diferentes ocasiones.

Es en la etapa de ejecución donde entra en juego nuestra herramienta, *GAmeraHOMggen*. Una vez ejecutados los casos de prueba contra todos los mutantes y el programa original y calculado el fitness de los individuos, se volverán a generar nuevos casos de prueba a partir de los anteriores, a priori más adaptados, a través del proceso marcado por nuestro algoritmo genético hasta que se verifique alguna de las condiciones de terminación definidas.

Existe también la posibilidad de ejecutar los casos de prueba no contra todos los mutantes, sino sólo contra algunos de ellos. Deberemos definir los mutantes sobre los que ejecutar los diferentes casos de prueba en el fichero de configuración. En el caso de que no se definan, se entenderá que se quieren ejecutar contra todos.

El resto de componentes explicados funcionan de la misma forma, por lo que nuestra herramienta se integra perfectamente en el sistema ya existente añadiendo una nueva funcionalidad.

Para la ejecución del programa original y los mutantes contra los casos de prueba, *GAmeraHOM-ggen* emplea el motor WS-BPEL 2.0, ActiveBPEL 4.1 [17] y BPELUnit [18].

5.1.1. Entorno de los productos

El producto desarrollado se compone de una interfaz y la implementación de la misma presentes en los paquetes *gamerahom-ggen/gamera/ggenAPI* y *gamerahom-ggen/gamera/ggen* respectivamente.

El componente correspondiente al análisis, generador y ejecución de mutantes se encuentra en *gamerahom-api* y su implementación en *gamerahom-core* y *gamerahom-bpel*.

Por otra parte, el generador de datos aleatorio que se usará para generar la primera población se encuentra en *test-generator-api* y su implementación en *test-generator*.

5.2. Detalles de implementación

Una vez que hemos situado la herramienta dentro de los componentes ya existentes pasamos a mostrar el proceso a seguir por el algoritmo genético de *GAmeraHOm-ggen* con el diagrama de actividades de la figura 5.3

El primer paso que tendremos que realizar es generar la primera población de individuos, llamada población inicial, compuesta por individuos generados de manera aleatoria, respetando las restricciones y los tipos de datos impuestos por cada composición. Una vez realizado esto calcularemos el fitness de cada uno de ellos antes de optimizar la población.

Comprobaremos si se cumple alguna de las condiciones de parada configuradas, como son alcanzar un determinado número de generaciones, superar un porcentaje dado de mutantes muertos, estancamiento de la población... si alguna de ellas se cumple daremos por concluido el proceso.

En caso negativo iniciaremos la optimización de los individuos de la población. Un porcentaje de individuos de la nueva generación volverán a ser generados simplemente de manera aleatoria, sin tener en cuenta la generación anterior.

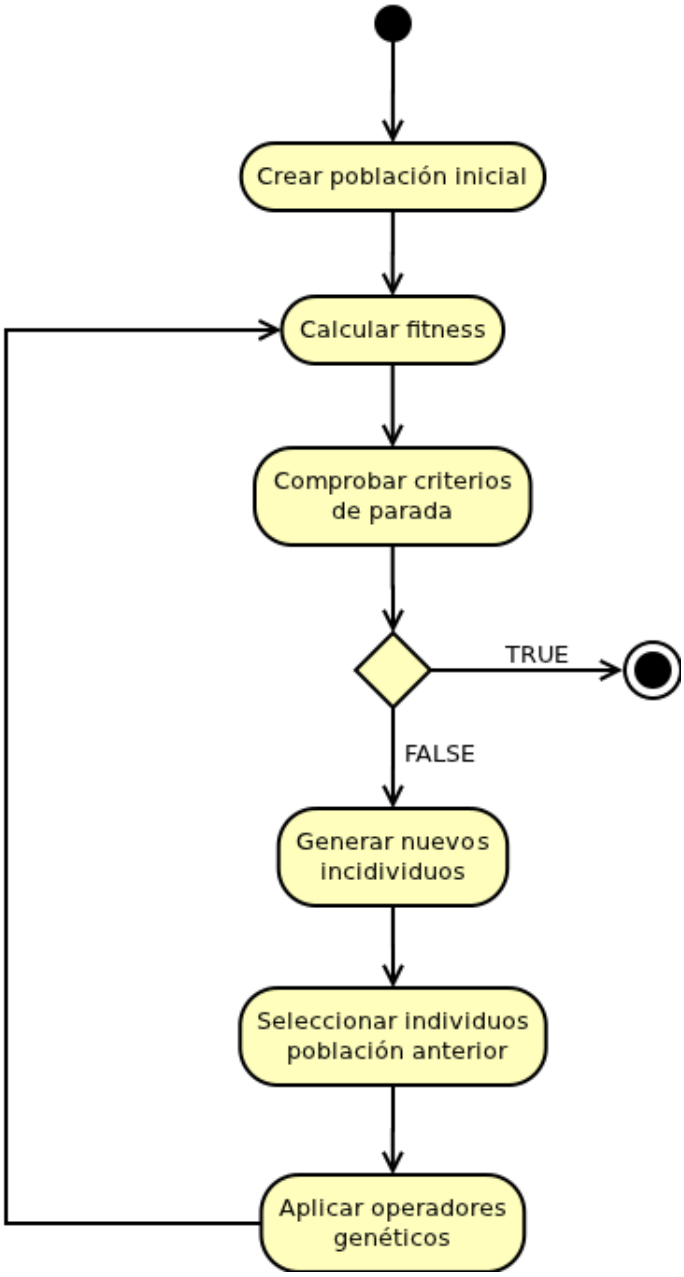

Figura 5.3.: Diagrama actividades

Del porcentaje restante hasta completar la población seleccionaremos los individuos que los diferentes métodos de selección implementados elijan. Estos son selección por ruleta y selección aleatoria.

A los individuos que hayan sido seleccionados en el paso anterior, se les aplicarán los operadores genéticos, el cruce y la mutación, para obtener otros más adaptados al medio. De esta forma, se completa la generación siguiente.

A continuación, se vuelve a repetir el proceso completo desde el cálculo del fitness hasta que se verifiquen algunas de las condiciones de terminación establecidas.

5.3. Diagrama de clases

En el diagrama de clases se describen las estructuras de datos de un sistema y las relaciones estáticas que existen entre ellas.

La diferencia fundamental entre el diagrama de clases y el modelo conceptual es que este diagrama no muestra gráficamente conceptos del mundo real, sino que describe los componentes software existentes.

5.3.1. Lanzador del algoritmo genético

La clase *Launcher* será la encargada de lanzar *GAmeraHOM-ggen* desde la línea de comandos. Para una correcta ejecución necesitará conocer la ruta del fichero de configuración YAML. En caso de que la ruta no sea la correcta se informará adecuadamente del error.

Gracias a la introspección toda la información contenida en el fichero de configuración será depositada en la clase *Configuration*, que tendrá todas las condiciones y parámetros necesarios para configurar el algoritmo.

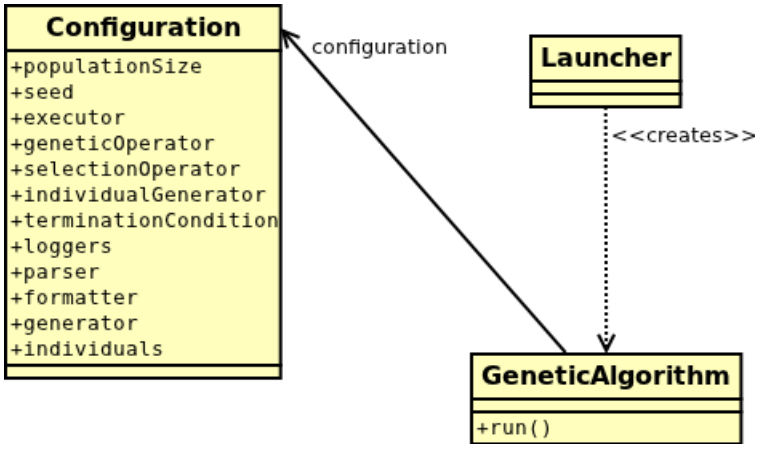

Figura 5.4.: Diagrama de clases del lanzador del algoritmo genético

Podemos ver un ejemplo de un fichero concreto en el listado 5.1.

Listado 5.1: Fichero de configuración YAML

```
1  populationSize: 6
2  seed: 42
3
4  executor: !!gamera.exec.BPELExecutor
5    testSuite: src/test/resources/LoanApprovalRPC/
6         loanApprovalProcess-velocity.bpts
7    originalProgram: src/test/resources/LoanApprovalRPC/
8         loanApprovalProcess.bpel
9    outputFile: target/loanApprovalProcess.bpel.out
10
11 geneticOperators:
12   - !!gamera.ggen.genetic.CrossoverOperator {probability: 0.4}
13   - !!gamera.ggen.genetic.MutationOperator {constantMutation: 10,
14                                             probability: 0.6}
15
16 individualGenerators:
17   !!gamera.ggen.generate.UniformGenerator {} : {percent: 0.2}
18
```

```
19 selectionOperators:
20   !!gamera.ggen.select.UniformRandomSelection {} : {percent: 0.6}
21   !!gamera.ggen.select.RouletteSelection {} : {percent: 0.4}
22
23 terminationConditions:
24   - !!gamera.ggen.term.PercentAllMutantsCondition {percent: 0.8}
25   - !!gamera.ggen.term.GenerationCountCondition {count: 3}
26   - !!gamera.ggen.term.StagnationMaximumFitness {count: 3}
27   - !!gamera.ggen.term.StagnationAverageFitness {count: 3}
28
29 loggers:
30   - !!gamera.ggen.log.MessageLogger {console: true, file: }
31   - !!gamera.ggen.log.HofLogger {console: false, file: hof.txt}
32
33 parser:
34   - !!testgen.parsers.spec.SpecParser {spec: src/test/resources/
35                                LoanApprovalRPC/data.spec}
36
37 formatter: !!testgen.formatters.VelocityFormatter {}
38
39 generator: !!testgen.generators.UniformRandomGenerator {}
40
41 individuals:
42   - [4,1,3]
43   - [7,1,3]
44   - [11,1,1]
45   - [13,2,1]
46   - [16,2,1]
```

Definiendo de esta manera tan sencilla los parámetros de configuración, a la hora de expandir el sistema, crear un nuevo operador genético, un nuevo generador de nuevos individuos... simplemente tendremos que añadirlo en la sección correspondiente e implementar la nueva clase y pasará a formar parte de nuestro sistema sin necesidad de

cambiar el resto de componentes.

1. En primer lugar podemos observar dos parámetros individuales que debemos configurar, como son el tamaño de la población y la semilla compartida en toda la herramienta que usaremos para generar números pseudoaleatorios. De esta forma conseguiremos tener resultados que se repitan con más frecuencia que facilitarán estudios estadísticos posteriores.

2. A continuación, como vemos en 6.1, se definirá el executor encargado de preparar el entorno, limpiar, generar y comparar los mutantes contra el programa original. Deberemos especificar la ubicación del conjunto de casos de prueba, la ruta del programa original y de un fichero de salida.

Listado 5.2: Definición del ejecutor usado

```
executor: !!gamera.exec.BPELExecutor
 testSuite: src/test/resources/LoanApprovalRPC/
      loanApprovalProcess-velocity.bpts
 originalProgram: src/test/resources/LoanApprovalRPC/
      loanApprovalProcess.bpel
 outputFile: target/loanApprovalProcess.bpel.out
```

3. Es momento de definir los operadores genéticos a usar en 5.3. Definiremos a su vez la probabilidad de aplicar el operador y una constante de mutación en el caso del operador de mutación.

Listado 5.3: Definición de los operadores genéticos

```
geneticOperators:
 - !!gamera.ggen.genetic.CrossoverOperator {probability: 0.4}
 - !!gamera.ggen.genetic.MutationOperator {constantMutation: 10,
      probability: 0.6}
```

4. En el listado 5.4 vemos la definición de los generadores de nuevos individuos usados. Junto a él, definiremos un valor que determinará la probabilidad de generar nuevos individuos usando dicho operador.

Listado 5.4: Definición de los generadores de individuos

```
1 individualGenerators:
2   !!gamera.ggen.generate.UniformGenerator {} : {percent: 0.2}
```

5. Lo mismo ocurre con los operadores de selección (ver 5.5), definiremos cada uno de los operadores de selección a usar, el método de la ruleta y otro aleatorio en este caso, y la probabilidad de aplicar cada uno de los operadores sobre los individuos de la población.

Listado 5.5: Definición de los operadores de seleccion

```
1 selectionOperators:
2   !!gamera.ggen.select.UniformRandomSelection {} : {percent: 0.6}
3   !!gamera.ggen.select.RouletteSelection {} : {percent: 0.4}
```

6. Para las condiciones de terminación (ver 5.6), definiremos un porcentaje en el caso de tratarse de la condición sobre el porcentaje de mutantes muertos o un valor de cuenta para el resto que nos indicará bien el número de generaciones máximas posible, o el número de generaciones que puede estar sin mejorar o bien el fitness del mejor individuo o el fitness medio de todos ellos.

Listado 5.6: Definición de las condiciones de parada

```
1 terminationConditions:
2   - !!gamera.ggen.term.PercentAllMutantsCondition {percent: 0.8}
3   - !!gamera.ggen.term.GenerationCountCondition {count: 3}
4   - !!gamera.ggen.term.StagnationMaximumFitness {count: 3}
```

```
5   - !!gamera.ggen.term.StagnationAverageFitness {count: 3}
```

7. El sistema tendrá dos posibles maneras de mantener registros (5.7). Con la primera mostraremos mensajes simples, aquello que vaya ocurriendo durante la ejecucuión del algoritmo. Algunos de ellos serán se empieza a analizar el programa original, se comparan los resultados de las salidas del programa original y los mutantes, se ha generado una nueva población... Con la segunda llevaremos el control de todos los individuos diferentes que hayan entrado en juego durante la ejecución del algoritmo, con el objetivo de terminar creando por ejemplo un fichero donde poder visualizar todos ellos.

   Podemos configurar que estos mensajes sean mostrados bien por consola o bien en un fichero externo. No es posible mostrarlos mediante las dos formas.

Listado 5.7: Definición de los loggers del sistema

```
1 loggers:
2   - !!gamera.ggen.log.MessageLogger {console: true, file: }
3   - !!gamera.ggen.log.HofLogger {console: false, file: hof.txt}
```

8. Configuraremos también los parámetros usados de *TestGenerator*(ver 5.9). El parser nos informará de la ruta donde se encuentra el fichero .spec en donde definiremos los tipos y sus restricciones; el formatter nos indica en qué formato se generarán los datos de cada población; y por último configuraremos el generador de tipos a usar. En estos momentos sólo existe el generador uniforme, pero en un futuro se implementarán también otros tipos de generadores basados en otras distribuciones de probabilidad, que podremos incluir en nuestra herramienta fácilmente.

Listado 5.8: Definición de los parámetros usados de *TestGenerator*

```
1 parser:
```

```
2   - !!testgen.parsers.spec.SpecParser {spec: src/test/resources/
3                                 LoanApprovalRPC/data.spec}
4
5 formatter: !!testgen.formatters.VelocityFormatter {}
6
7 generator: !!testgen.generators.UniformRandomGenerator {}
```

9. Por último, tenemos la opción de elegir los mutantes concretos sobre los que ejecutar el conjunto de casos de prueba generados. En caso de no especificar esta opción, se ejecutarán los casos de prueba contra todos los mutantes.

   Deberemos respetar el formato con el que definiremos los mutantes correspondiéndose a la estructura mencionada en 5.2. Cada fila se corresponderá con un mutante, coincidiendo el primer parámetro con el operador, el segundo con la instrucción y el tercero con el atributo.

   Comprobaremos, una vez analizado el programa orignal, que los mutantes que hemos definido existen realmente. Si llamamos O al operador, I a la instrucción y A al atributo ([O, I, A]), estos parámetros deberán respetar las siguientes condiciones:

   - $1 \leq O \leq$ nº máximo operadores
   - $1 \leq I \leq$ nº de instrucciones del operador O
   - $1 \leq A \leq$ nº de atributos del operador O

Listado 5.9: Definición de los mutantes

```
1 individuals:
2   - [4,1,3]
3   - [7,1,3]
4   - [11,1,1]
5   - [13,2,1]
6   - [16,2,1]
```

5.3.2. Ejecución de los individuos

Tras haber cargado todos los parámetros necesarios para el correcto funcionamiento de la herramienta, debemos calcular el fitness de los individuos. Como comentamos en la sección 5.1 la parte encargada de esto se encuentra en *gameraHOM-api*.

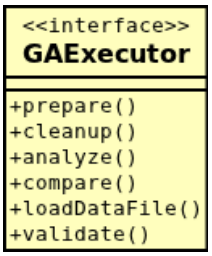

Figura 5.5.: Diagrama de clases del executor

En la figura 5.5, se define la interfaz con los métodos que necesitamos para preparar el entorno (`prepare()`) y analizar el programa original (`analyze()`). Con el método `loadDataFile()` podemos cargar el fichero con extensión VM (Apache Velocity) en donde se encontrarán los datos generados para rellenar los distintos casos de prueba de cada composición.

Por último el método `compare()` nos permite comparar los resultados de ejecutar los diferentes casos de prueba contra el programa original y sus mutantes para posteriormente calcular el fitness de los individuos.

Estos resultados se vuelcan en la clase *ComparisonResults*. En la figura 5.6 observamos la estructura de dicha clase. Con un enumerado definiremos los posibles valores de salida tras ejecutar el programa contra los casos de prueba.

Como hemos venido comentando hasta ahora, existen 3 resultados posibles que indican para cada mutante y caso de prueba si existe diferencia o no con respecto el programa original.

- $t_{ij} = 0$ Si el mutante i no es matado por el caso de prueba j (SAME_OUTPUT).
- $t_{ij} = 1$, si el mutante i es matado por el caso de prueba j (DIFFERENT_OUTPUT).

- $t_{ij} = 2$, si el mutante i produce un error en la ejecución sobre alguno de los casos de prueba (INVALID).

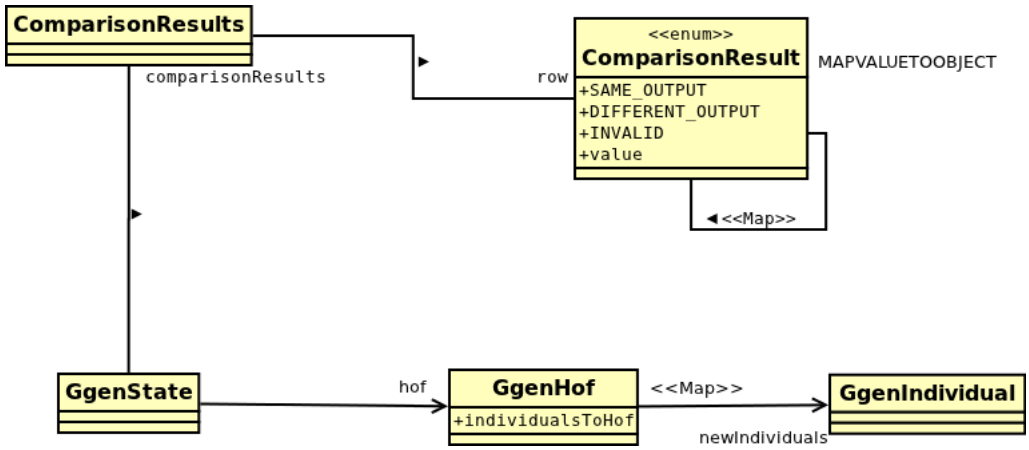

Figura 5.6.: Diagrama de clases del análisis de los individuos

Podemos ver también como aparecen tres clases nuevas más *GgenState*, *GgenHof* y *GgenIndividual*. La primera mantendrá información útil de las distintas generaciones por las que pasará el algoritmo, como por ejemplo toda la relativa a la información que necesitan consultar las condiciones de parada para ver si se satisfacen sus premisas o no como los resultados tras la ejecución de los individuos, el número de generación actual... La segunda, en cambio, llevará un control de los nuevos individuos que se vayan generando para posteriormente añadirlos a un fichero de registros del sistema. Y por último *GgenIndividual*, los individuos que generaremos en sí y que explicaremos con detalle en 5.3.3

5.3.3. Representación de los individuos

Cada individuo codifica los valores que tomarán los casos de prueba, es decir, cada individuo será la representación de uno de los casos de prueba. Se añade también un valor más que indicará el fitness o aptitud de los individuos.

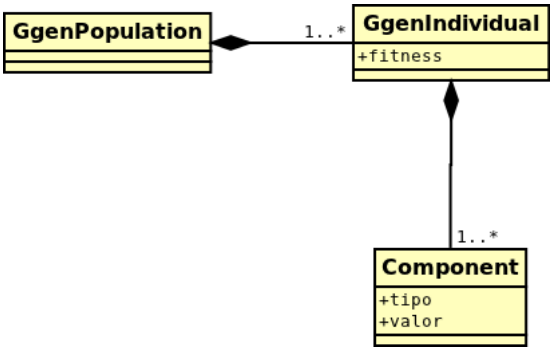

Figura 5.7.: Diagrama de clases del individuo y su población

Estructura interna del individuo

Un individuo estará formado por un conjunto de componentes. Cada componente nos indicará el tipo del que se trata y el valor concreto que tomará. De forma que si por ejemplo, un caso de prueba consiste en una cadena que represente el nombre de una persona y un valor que indique la edad de la misma, nuestro individuo estará formado por dos componentes: el primero nos informará que se trata de un componente tipo cadena seguido del nombre de la persona, mientras que el segundo nos indicará que se trata de un valor entero acompañado de la edad correspondiente también.

El resto de individuos que compongan la población estarán codificados de la misma forma cambiando únicamente el valor de cada tipo.

Fitness o aptitud

Es un valor que nos informará de cuánto de bueno es un individuo.

La aptitud de un individuo [19] va a ser función del número de mutantes muertos. Sin embargo, hemos de tener en cuenta que si empleamos como aptitud únicamente el número de mutantes muertos, aquellos casos de prueba que maten a un único mutante y éste sólo sea matado por ese caso de prueba, obtendrá poca aptitud. Por este motivo, para evaluar la aptitud de los individuos debemos tener en cuenta un factor adicional, el número de casos de prueba que matan al mismo mutante. Así, si M representa el número

máximo de mutantes y *T* el número máximo de casos de prueba de los que disponemos, la aptitud de un individuo será:

$$Fitness(I) = \sum_{j=1}^{M} \frac{t_{ij}}{\sum_{i=1}^{T} t_{ij}}$$

Siendo I el individuo del que queremos saber su fitness, M el número total de mutantes, T el número total de casos de prueba y t_{ij} el resultado de la comparación del mutante i con el individuo o caso de prueba j. Habrá que tener en cuenta, que si el resultado de la prueba es inválido, el valor de la aptitud del mutante será cero.

Ejemplo

A continuación expondremos un ejemplo clarificativo de cálculo del fitness de 4 individuos dada una matriz de ejecución.

Sea E la matriz de ejecución correspondiente donde las filas representan los diferentes mutantes y las columnas cada uno de los casos de prueba o individuos:

$$E = (MxT) = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 2 & 2 & 2 & 2 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Las aptitudes de los 4 individuos son:

$$F(I_1) = 1/3 + 1/2 + 0 + 0/2 = 5/6$$

$$F(I_2) = 0/3 + 0/2 + 0 + 0/2 = 0/6$$

$$F(I_3) = 1/3 + 0/2 + 0 + 1/2 = 5/6$$

$$F(I_4) = 1/3 + 1/2 + 0 + 1/2 = 8/6$$

5.3.4. Generadores de individuos

El generador de indivuos se encarga de crear nuevos individuos. Estos nuevos individuos no partirán de características de generaciones anteriores, sino que serán generados siguiendo las directrices marcadas en el fichero de configuración.

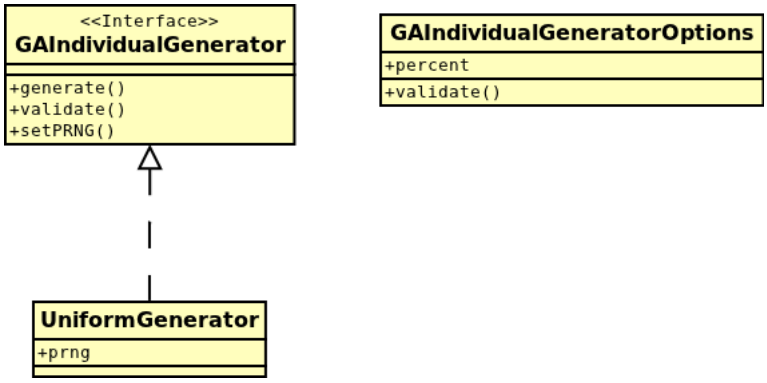

Figura 5.8.: Diagrama de clases de los generadores de individuos

En estos momentos el único generador disponible es el generador uniforme de individuos, pero podríamos crear otros en base a otros tipos de distribuciones estadísticas u otros factores.

Aprovechando la estructura que nos permite YAML, definiremos la estructura de la siguiente forma:

```
  individualGenerators: !!gamera.ggen.generate.NombreClase
OPCIONES_INTERNAS : OPCIONES_COMUNES
```

Las opciones comunes son implementadas en una clase común a todos los generadores, en concreto en *GAIndividualGeneratorOptions*, mientras que las opciones internas se implementan en cada clase concreta que implemete a la interfaz *GAIndividualGenerator*.

En este caso, la única necesaria será la probabilidad de aplicar dicho operador, que se corresponde con una opción común que habría que configurar para cada uno de los generadores que se creasen.

La suma de las probabilidades de todos los generadores, en este caso la única existente, deberá ser menor o igual a 1, e indicará dado un tamaño de población, el porcentaje

de individuos nuevos que será creado en cada generación aplicando dicho método.

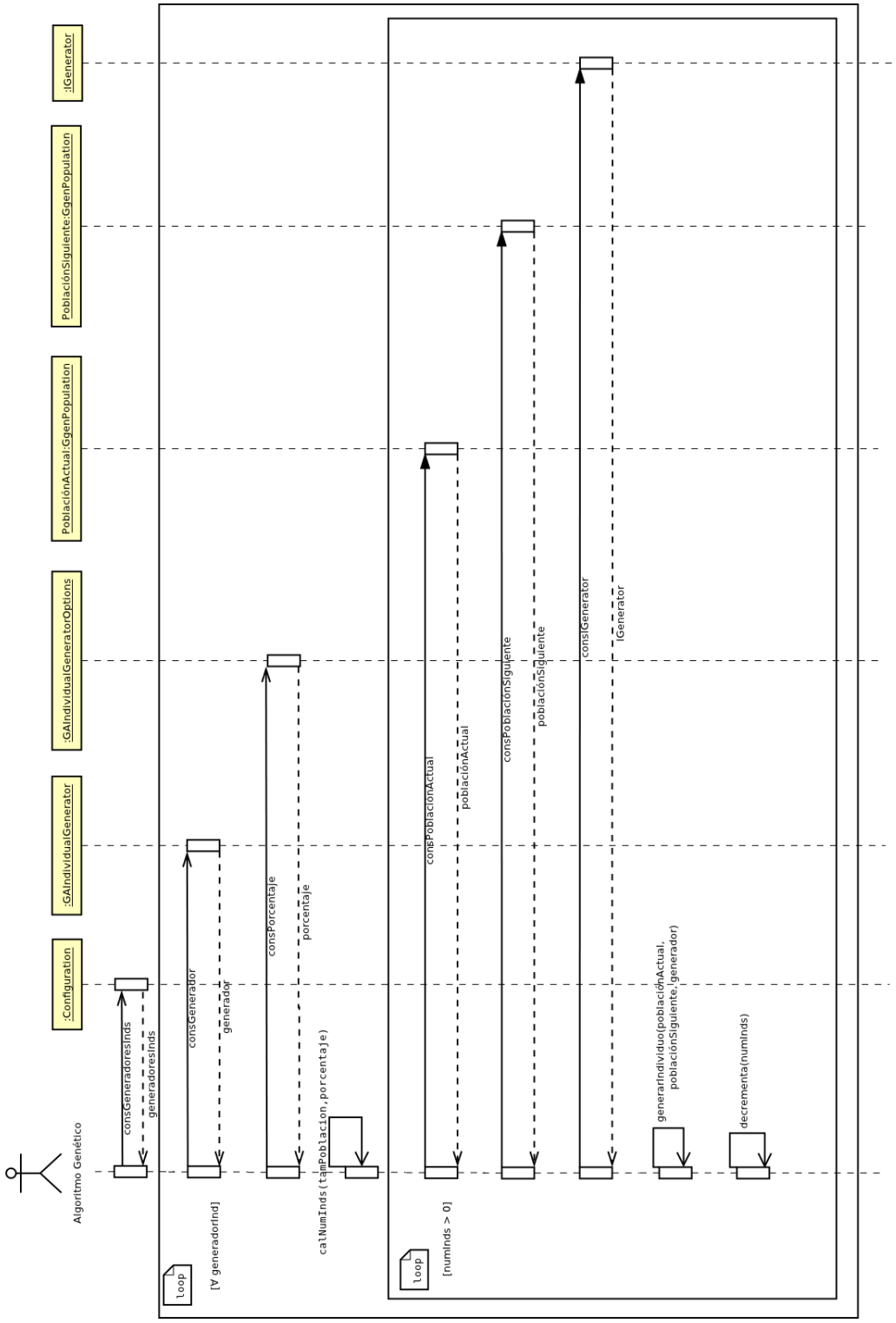

Figura 5.9.: Diagrama de secuencia del generador de individuos

5.3.5. Selección de individuos

La selección de individuos indicará las diferentes opciones que tendremos para seleccionar los individuos de una población para posteriormente aplicar sobre los individuos seleccionados, operadores genéticos.

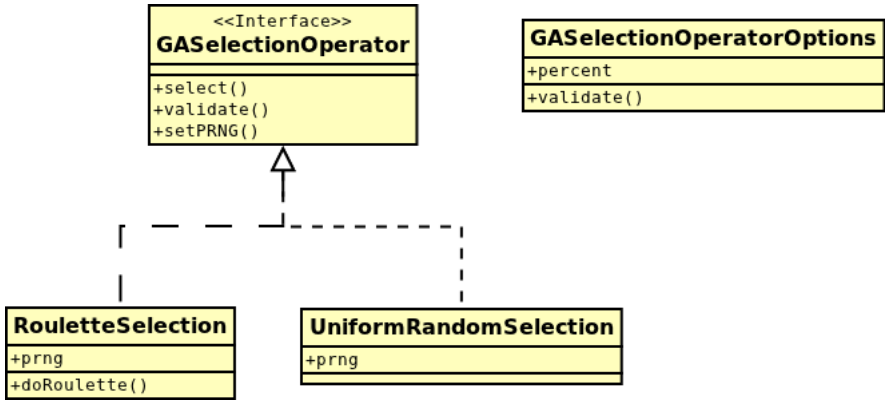

Figura 5.10.: Diagrama de clases de los operadores de selección

Al igual que ocurría con los generadores se seguirá la misma estructura para definir los operadores de selección en YAML.

```
  selectionOperators:!!gamera.ggen.select.NombreClase
OPCIONES_INTERNAS : OPCIONES_COMUNES
```

Definiremos un valor en opciones comunes que nos indicará la probabilidad de aplicar cada operador de selección, es decir, dado un tamaño de población, el porcentaje de individuos a seleccionar por cada método. Este porcentaje de individuos no será con respecto al tamaño de población, sino con respecto a los individuos que falten para completar la población tras haber aplicado el generador de nuevos individuos.

Para los operadores implementados ya, selección por ruleta y aleatoria, no son necesarios configurar ninguna opción específica interna, simplemente la probabilidad ya comentada.

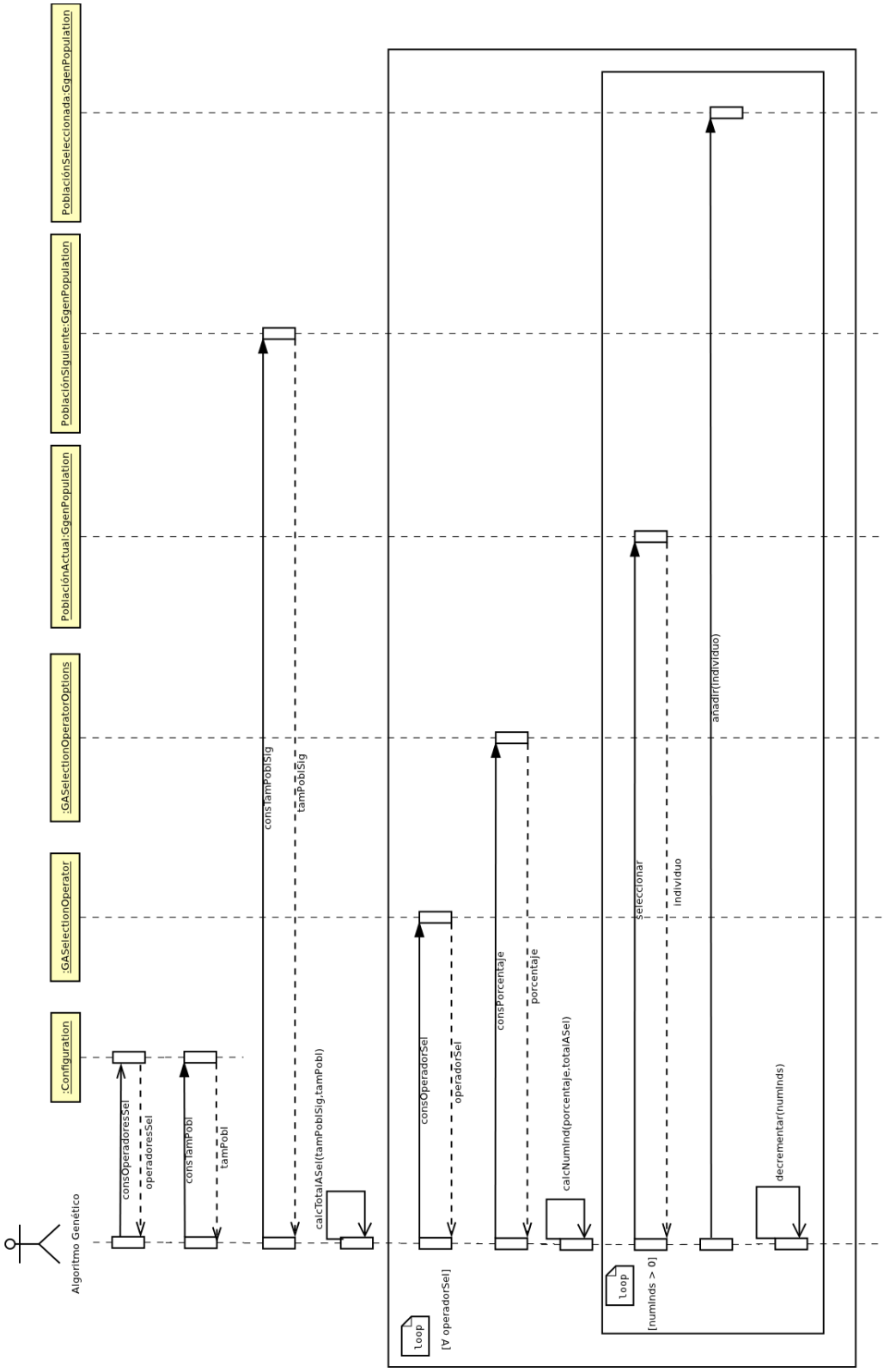

Figura 5.11.: Diagrama de secuencia del operador de selección de individuos

5.3.6. Operadores genéticos

Los operadores genéticos generarán descendientes a partir de unos individuos previamente seleccionados con el objetivo de crear a partir de estos últimos, individuos más adaptados al medio.

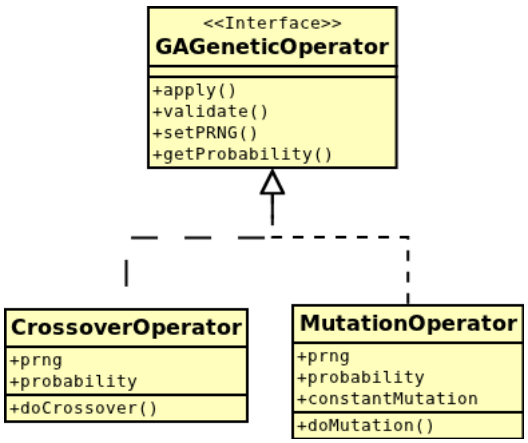

Figura 5.12.: Diagrama de clases de los operadores genéticos

Esta es la estructura que habrá que seguir para definir los operadores genéticos, incluyendo en `OPCIONES` las opciones específicas de cada operador que se comentarán posteriormente.

```
!!gamera.ggen.genetic.NombreClase OPCIONES
```

Los operadores que se han diseñado son la mutación y el cruce. La mutación consiste en modificar el valor de un solo componente del individuo padre creando un nuevo hijo; el cruce, por contra, intercambia características de dos individuos progenitores, generando otros dos nuevos individuos llamados hijos.

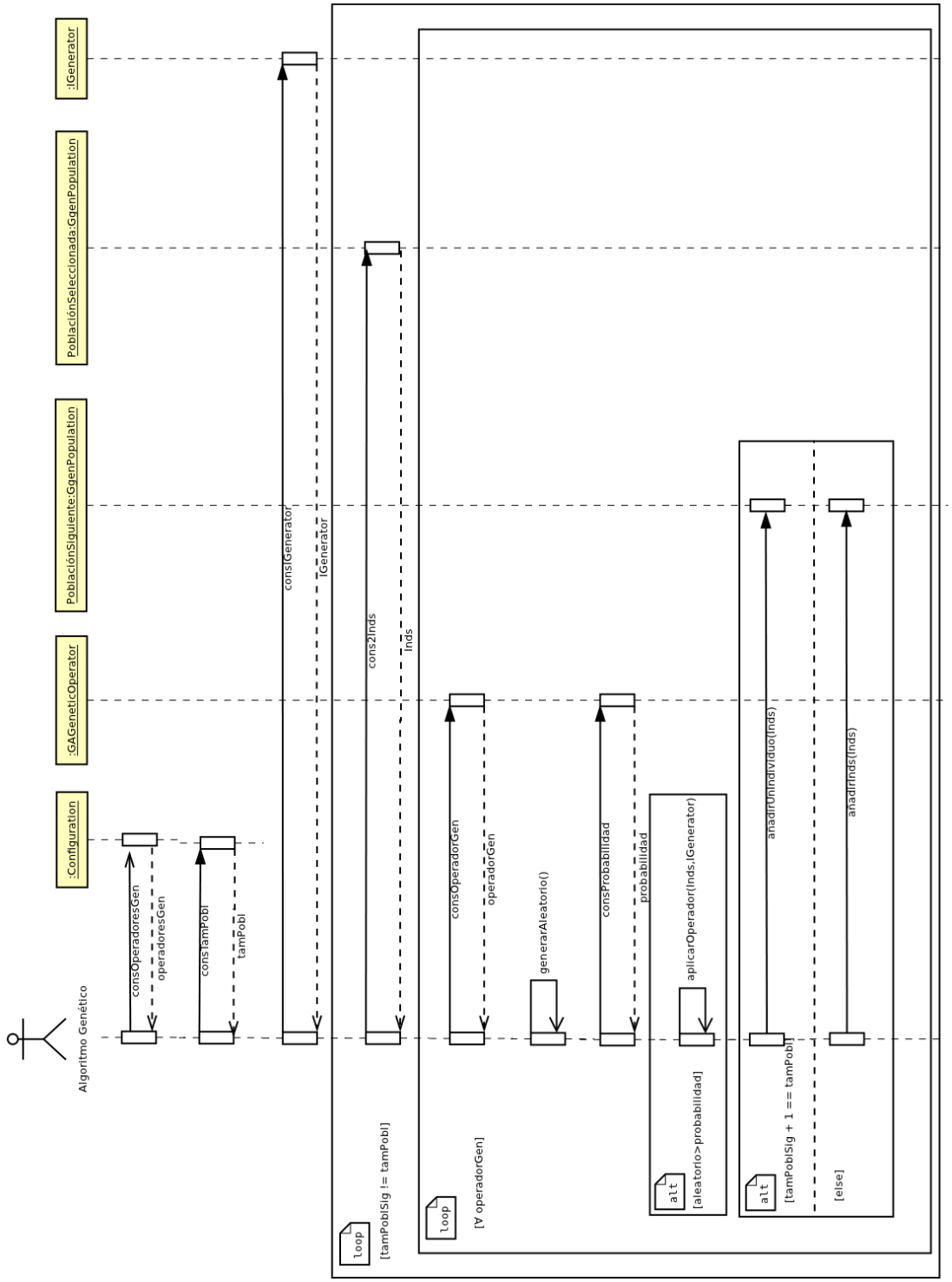

Figura 5.13.: Diagrama de secuencia del operador genético de individuos

Cruce

Se generará un valor aleatorio entre 0 y 1 que determinará si se aplica el cruce o no en función de si este valor es mayor o menor que la probabilidad de cruce definida en el fichero de configuración.

De no aplicarse, los hijos serán una copia idéntica de sus progenitores, en cambio si se aplica, habrá que determinar el punto a partir del cual se produzca el cruce de ambos individuos, siendo escogido este de forma aleatoria entre todos los componentes del individuo.

Mutación

En la mutación deberemos definir además de la probabilidad de aplicar dicho operador, una constante de mutación. Esta constante se usará para determinar el valor que tomen algunos de los casos de prueba una vez mutados.

Al igual que ocurre con el cruce, se generará un valor aleatorio entre 0 y 1 que determinará en función de la probabilidad definida si aplicaremos o no el operador de mutación. En caso de aplicarlo, se seleccionará también de forma aleatoria la posición del individuo a mutar (elegiremos un componente de los que conforman un individuo), variando la manera de realizar la mutación en función de los parámetros definidos en YAML como sigue:

- Si el tipo de la posición elegida se trata de un entero o un flotante:

$$valorNuevo = valorantiguo \pm tamMutation$$

$$tamMutation = 1/probability * constantMutation * Random(0-1)$$

- Si el tipo de la posición elegida se trata de una cadena: cambiaremos el valor de la cadena por otra de las posibilidades existentes según las restricciones definidas en el fichero *spec*.

- Si el tipo de la posición elegida es una lista, cambiaremos el tamaño de la misma de la siguiente forma:

$$valorNuevo = 1/probability * numElements * Random(0 - 1)$$

- Por último, si se trata de de una tupla, no realizaremos ninguna operación, y pasaremos al siguiente individuo.

5.3.7. Criterios de parada

Los criterios de parada son aquellas condiciones que el algoritmo comprobará siempre que una generación ha sido creada y el fitness de sus individuos calculado. Cuando alguno de los criterios se cumpla, el algoritmo detendrá su ejecución. Mientras no se cumplan, se seguirá con la creación de una nueva generación.

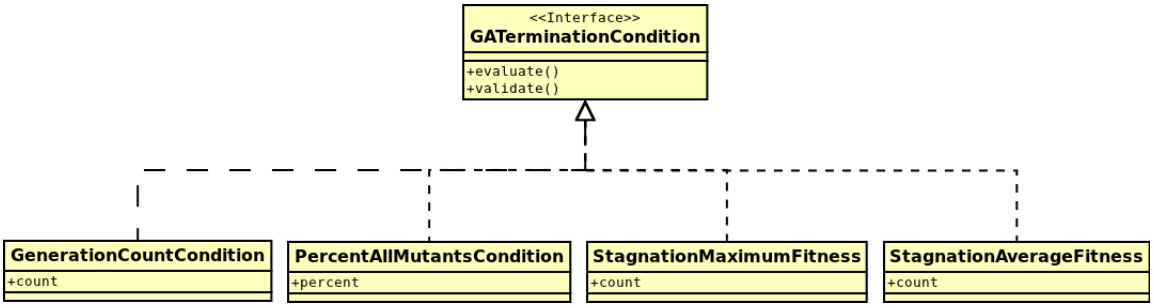

Figura 5.14.: Diagrama de clases de las condiciones de terminación

Cada criterio de parada se define en el fichero de configuración YAML como sigue siendo `OPCIONES` el porcentaje o el valor de cuenta que se explica a continuación:

```
- !!gamera.ggen.term.NombreClase OPCIONES
```

Pasamos a describir cada una de las condiciones de terminación:

Contador de generaciones

Comprueba si se ha llegado a un determinado número de generaciones, especificado en las opciones del operador, en cuyo caso, se detiene la ejecución.

Matar a un porcentaje de mutantes

Cuando calculamos el fitness de una generación, podemos ver cuantos mutantes estamos matando con los casos de prueba generados. Si la división entre el número de mutantes muertos y el total es igual al valor dado, se finaliza el algoritmo.

Se considera que un mutante está muerto si para alguno de los casos de prueba para los que se ejecuta produce una salida diferente con respecto al programa original.

Estancamiento en la evolución del fitness máximo

El fitness del mejor individuo de la población debería ir mejorando tras las sucesivas generaciones. Pero puede darse el caso de que empiece a empeorar o se establezca en un valor sin conseguir mejora. Si de forma consecutiva no mejora durante el número de generaciones que se fije, el algoritmo detendrá su ejecución, pues asumiremos que ya hemos encontrado los mejores casos de prueba posibles y no pueden ser mejorados más.

Estancamiento en la evolución del fitness medio

Sigue la misma lógica que el caso anterior, pero lo que deberá ir mejorando será en este caso, el fitness medio de los individuos.

5.3.8. Loggers

Los loggers son registros que informan de lo que está ocurriendo en el sistema, el algoritmo genético en este caso.

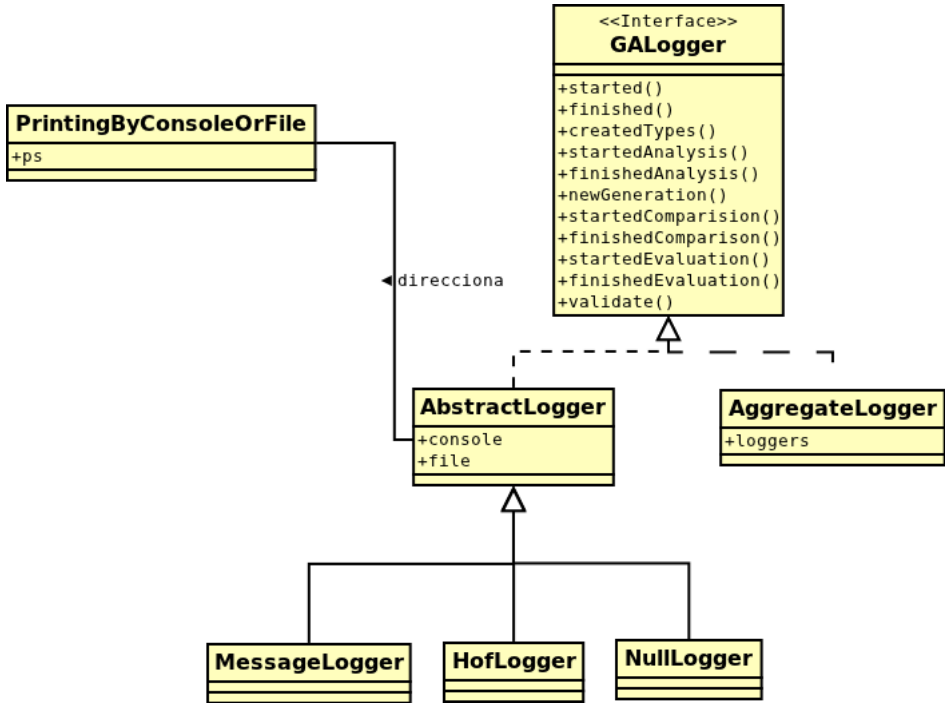

Figura 5.15.: Diagrama de clases de los loggers

Cada logger se define en el fichero de configuración así:

- `!!gamera.ggen.log.NombreClase OPCIONES`

Las opciones que necesitamos definir serán un booleano que nos informe si queremos que muestre información de un logger concreto por consola, `console`, y el nombre del fichero en donde queremos generar los logs en el caso de que queremos que se muestren en un fichero. Existe la restricción de que para un mismo logger no se permite la salida por consola y por fichero.

Se definen tres loggers distintos, uno de mensajes simples `MessageLogger` y otro que mostrará los diferentes individuos que se hayan generado durante la completa ejecución del algoritmo, *HofLogger* (el salón de la fama). Además existe un logger vacío que entrará en juego cuando no se defina ninguno de los dos anteriores *NullLogger*.

La clase *PrintingByConsoleOrFile* se encargará de direccionar el flujo de mensajes según los parámetros especificados hacia la consola o el fichero.

Existe también la clase *AggregateLogger* que reúne todos los loggers del sistema para que el algoritmo pueda trabajar con ellos de forma transparente.

5.3.9. Componentes usados de *TestGenerator*

A continuación, explicaremos los componentes de *TestGenerator* que usaremos en nuestra herramienta.

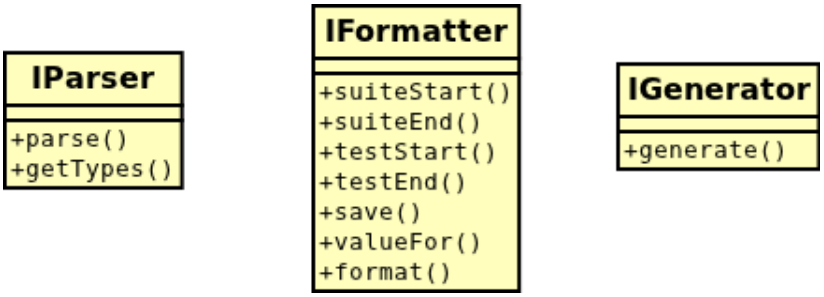

Figura 5.16.: Clases usadas de TestGenerator

En primer lugar especificaremos el analizador (parser) a utilizar siguiendo esta estructura:

- `!!testgen.parsers.NombreClase OPCIONES`

La clase donde se define el analizador es *SpecParser* y además como opciones a usar deberemos especificar la ruta del fichero SPEC. El fichero SPEC es donde se definen los tipos de datos usar y las restricciones existentes para cada uno de ellos.

Cuando hablamos de formatter nos referimos a la forma en la que presentaremos los datos de salida. En nuestro caso, usaremos la opción implementada en *TestGenerator* para presentar los datos en formato Velocity. Será el formato necesario para que el fichero BPTS, en donde se definen los casos de prueba, pueda cargar el valor de los distintos datos a utilizar.

Por último, tendremos que definir el generador de datos a usar. Por el momento sólo existe el generador de datos uniforme, por lo que será el que utilicemos para realizar dicha tarea.

Para ninguno de estos dos últimos casos son necesarios indicar opciones, se deja la estructura preparada por si en un futuro se implementa otro tipo de generador o formateador que sí lo requiera poder hacerlo sin mayor dificultad.

```
formatter: !!testgen.formatters.NombreClase OPCIONES
generator: !!testgen.generators.NombreClase OPCIONES
```

CAPÍTULO 6

IMPLEMENTACIÓN Y PRUEBAS

Trataremos de explicar en este capítulo las herramientas empleadas en el desarrollo del proyecto y la razón de su uso. Explicaremos también el tipo de pruebas realizadas para verificar la validez de la solución.

6.1. Tecnologías y lenguajes de programación usados para la implementación

6.1.1. Java

El lenguaje utilizado para la realización de este proyecto ha sido Java [11]. La elección del lenguaje viene como requisito impuesto por el grupo. De esta forma podemos reutilizar código existente y así integrar la herramienta dentro de *GAmera* para que este proyecto pueda ser reutilizado también en el futuro.

Java es un lenguaje de programación de alto nivel orientado a objetos, desarrollado por James Gosling en 1995. El lenguaje en sí mismo toma mucha de su sintaxis de C, Cobol y Visual Basic, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. La memoria es gestionada mediante un recolector de basura.

Las ventajas aportadas por el paradigma de programación orientado a objetos son [20]:

- Cercanía de sus conceptos a los del mundo real.
- Proceso de desarrollo más sencillo y rápido.
- Facilita reutilización de diseño y códigos.
- Modificaciones, extensiones y adaptaciones más sencillas.
- Sistemas más estables y robustos.

La utilización de Java nos aporta también la ventaja de generar buena documentación de manera fácil y sencilla. Javadoc [21] es una utilidad ofrecidad por Oracle para generar documentación de APIs en formato HTML a partir de código fuente Java. Es el estándar de la industria para documentar clases Java.

Una ayuda inestimable ha sido también el uso del IDE Eclipse Indigo. Eclipse tiene un muy buen soporte de refactorización que nos permite renombrar variables, métodos, clases... de forma fácil, rápida y segura, así como ayuda a la codificación, sugiriendo como poder arreglar fallos de compilación. También nos ofrece numerosos asistentes para crear clases, tests...

6.1.2. YAML

YAML [12] es un formato de serialización de datos legible por humanos inspirado en lenguajes como XML, C, Python, Perl, así como el formato para correos electrónicos especificado por el RFC 2822. YAML fue propuesto por Clark Evans en 2001, quien lo diseñó junto a Ingy döt Net y Oren Ben-Kiki.

Se creó bajo la creencia de que todos los datos pueden ser representados adecuandamente como combinaciones de listas, maps y datos escalares, lo cual se ha podido comprobar que es verdad.

En principio no se conocía su existencia, pero ha sido de gran ayuda una vez conocido su funcionamiento y su potencial.

Los ficheros de configuración de *GAmeraHOM-ggen* están escritos siguiendo el formato YAML. Se integra a la perfección con Java gracias a la introspección [22]. Esta técnica nos permite cargar objetos de los que no se sabe nada, así como encontrar sus métodos, constructores...

Las características que presentan los ficheros YAML son:

- Los contenidos en YAML se describen utilizando el conjunto de caracteres imprimibles de Unicode, bien en UTF-8 o UTF-16.
- La estructura del documento se denota indentando con espacios en blanco; sin embargo no se permite el uso de caracteres de tabulación para indentar.
- Los miembros de las listas se denotan encabezados por un guión con un miembro por cada línea, o bien entre corchetes y separados por coma espacio.
- Los arrays asociativos se representan usando los dos puntos seguidos por un espacio en la forma çlave: valor", bien uno por línea o entre llaves y separados por coma seguida de espacio.
- Un valor de un array asociativo viene precedida por un signo de interrogación, lo que permite que se construyan claves complejas sin ambigüedad.
- Los valores sencillos (o escalares) por lo general aparecen sin entrecomillar, pero pueden incluirse entre comillas dobles, o comillas simples.
- En las comillas dobles, los caracteres espaciales se pueden representar con secuencias de escape similares a las del lenguaje de programación C, que comienzan con una barra invertida.
- Se pueden incluir mútliples documentos dentro de un único flujo, separándolos por tres guiones; los tres puntos indican el fin de un documento dentro de un flujo.
- Los nodos repetidos se pueden denotar con un ampersand y ser referidos posteriormente usando el asterisco.

- Los comentarios vienen encabezados por la almohadilla y continúan hasta el final de la línea.
- Los nodos pueden etiquetarse con un tipo o etiqueta utilizando el signo de exclamación seguido de una cadena que puede ser expandida en una URL.
- Los documentos YAML pueden ser precedidos por directivas compuestas por un signo de porcentaje seguidos de un nombre y parámetros delimitados por espacios. Hay definidas dos directivas en YAML 1.1:
  - La directiva %YAML se utiliza para identificar la versión de YAML en un documento dado.
  - La directiva %TAG se utiliza como atajo para los prefijos de URIs. Estos atajos pueden ser usados en las etiquetas de tipos de nodos.

Ejemplo práctico

A continuación explicaremos un ejemplo de uso muy simple. Definiremos en un fichero *yaml* dos parámetros, por ejemplo el nombre de una persona y la lista de la compra tal y como se observa.

Listado 6.1: Ejemplo yaml

```
1 nombre: Antonio Perez
2
3 compra:
4   - huevos
5   - leche
6   - pan
7   - aceite
```

Una vez que hemos leído el fichero de configuración, con tan sólo definir en una clase *Java*, que llamaremos *Configuracion* los atributos `String nombre` y `List<String>compra` junto con sus métodos *get* y *set*, tendremos sendos atributos, una vez ejecutado el programa, inicializados, con el valor que hayamos indicado en el

fichero. En este caso en el atributo nombre tendremos *Antonio Perez* y en compra una lista con *huevos, leche, pan y aceite*.

6.1.3. Maven

Maven [23] es una herramienta diseñada para la gestión y construcción de proyectos Java. Fue creada por Jason van Zyl, de Sonatype, en 2002. Su funcionalidad es parecida a *Apache Ant*. Inicialmente estaba dentro del proyecto Jakarta, pero actualmente es un proyecto de nivel superior de la Apache Software Foundation.

Usa un POM (Project Object Model) para describir el proyecto software a construir. En él se indican las dependencias con otros módulos, los componentes externos y el orden de construcción de los elementos. La diferencia fundamental con Apache Ant es que se escriben de forma declarativa, en vez de en forma imperativa; es decir, en vez de indicar cómo construir algo, se dice qué hay que construir, y las convenciones de Maven controlarán el proceso de construcción.

Maven está listo para usar en red, ya que su motor puede descargar dinámicamente los plugins de un repositorio. Dicho repositorio provee acceso a muchas versiones de diferentes proyectos como Open Source en Java, de Apache y de otras organizaciones. Este repositorio y su sucesor reorganizado, Maven 2, intentan ser el mecanismo por defecto de distribución de aplicaciones en Java, pero su adopción está siendo lenta. Maven permite subir artefactos al repositorio al final de la construcción de la aplicación, de forma que cualquier usuario tiene acceso a ella.

Usa una arquitectura basada en plugins que permite que utilice cualquier aplicación controlable a través de la entrada estándar. En teoría, esto permite que cualquiera pueda escribir sus propios plugins para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, etc. para cualquier otro lenguaje; pero en la realidad, Maven apenas soporta otros lenguajes distintos a Java.

Está construido alrededor de la idea de reutilización de la lógica de construcción: los proyectos se construyen generalmente con patrones similares; una elección lógica sería reutilizar los procesos de construcción. La idea no es reutilizar el código o funcionalidad,

sino cambiar la configuración del código escrito.

Los proyectos en Maven cuentan con una serie de etapas, llamadas ciclo de vida. Para pasar a una etapa, es necesario haber completado con éxito las etapas anteriores. Estas etapas representan las distintas fases por las que un proyecto software ha de pasar. Las etapas más importantes son:

1. *compile*: compila el código.
2. *test*: ejecuta las pruebas unitarias.
3. *package*: empaqueta el *bytecode* resultado de compilar en un *.jar*.
4. *install*: instala el *.jar* en el repositorio local de Maven.
5. *deploy*: despliega el *.jar* en el repositorio remoto de Maven.

6.2. Integración continua

6.2.1. Subversion

Subversion [24] es un sistema de control de versiones. Fue diseñado para reemplazar a CVS. Entre las ventajas existentes de usar Subversión podemos destacar:

- Llevamos un historial de los cambios: cada vez que terminamos de trabajar en algo concreto o por un periodo de tiempo subimos nuestros cambios al repositorio y realizamos lo que se llama un `commit`. En el repositorio se almacena cada cambio como una *revisión* pudiendo volver a revisiones anteriores en cualquier momento. Esto garantiza el poder volver a versiones antiguas si se detectan fallos en funcionalidades donde antes no existían.
- Podemos colaborar con otras personas: al subir todos los cambios a una forja a la que podemos acceder a través de la red, podemos ver el trabajo realizado por otros miembros de ella y corregir o añadir lo que creamos conveniente. Además, como comentamos en el punto anterior, siempre su desarrollador principal podrá volver

a una revisión anterior si no está de acuerdo con los cambios, así como el resto de desarrolladores.

- Podemos trabajar en varias cosas diferentes: por la misma razón comentada anteriormente, al ir subiendo todos los cambios a una forja, podemos estar trabajando en distintos proyectos, y subir los cambios realizados en cada uno de ellos siempre que lo creamos oportuno de forma totalmente independiente al resto de proyectos en los que se trabaje.

- Hacemos copia de seguridad de todo el historial: lo que nos garantiza no perder nuestro trabajo. Igual que cuando trabajamos en local, siempre puede ocurrir un fallo en el sistema que nos haga perder la información, trabajando también con un servidor externo, las posibilidades de perder información importante se reducen.

- Maneja los ficheros binarios de forma eficiente.

- Permite el bloqueo de archivos para que no sean editados por más de una persona al mismo tiempo y asegurarnos la inexistencia de conflictos.

Todo nuestro proyecto, incluida la presente memoria, está subido a la forja del grupo UCASE usando Subversion. Se elige este sistema por el bajo coste que implica aprenderlo para todos los miembros del grupo, además de por ser el más extendido en todo tipo de proyectos en la actualidad.

6.2.2. Jenkins

Jenkins es un software de integración continua de código abierto escrito en Java, basado en el proyecto Hudson.

¿En qué consiste la integración continua? En hacer integraciones automáticas y periódicas de un proyecto para detectar posibles fallos. Es decir, Jenkins descarga, compila y ejecuta el proyecto y todos sus tests con la periocidad que se le indique con el objetivo de comprobar que todo funciona como debe.

En el caso del grupo UCASE se realizarán estas pruebas siempre justo cuando se hayan subido los nuevos cambios y una vez al día haya cambios nuevos o no. De esta forma nos aseguramos que el proyecto siempre funcione de manera correcta. Además, se puede configurar también para enviar correos a los desarrolladores cuando algo no funcione como debiera para que sea solucionado cuanto antes.

Podemos observar en 6.1 una captura del estado de todos los proyectos existentes actualmente en el grupo tras la última ejecución.

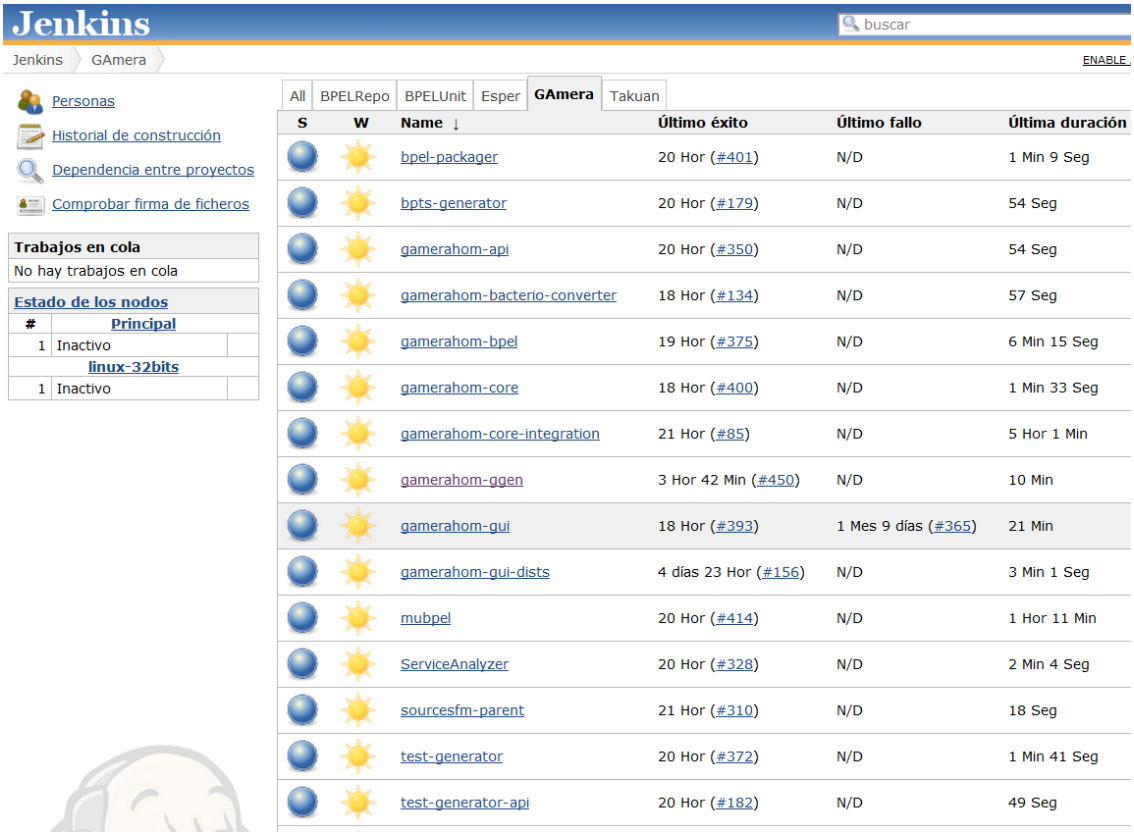

Figura 6.1.: Estado proyectos en Jenkins

El estado de los proyectos se encuentra en: https://neptuno.uca.es/jenkins/.

6.2.3. Sonar

Sonar [25] es una herramienta que nos permite controlar la calidad del código.

Sonar realiza varios análisis del código usando diferentes herramientas y nos informa de métricas sobre nuestro proyecto que nos serán de utilidad para saber los puntos débiles del mismo. A través de su interfaz intuitiva, nos presenta de forma unificada multitud de métricas: porcentaje de comentarios y de líneas duplicadas, complejidad de métodos y clases, porcentaje de cobertura con las pruebas unitarias así como los puntos para los que no se ha diseñado ninguna prueba...

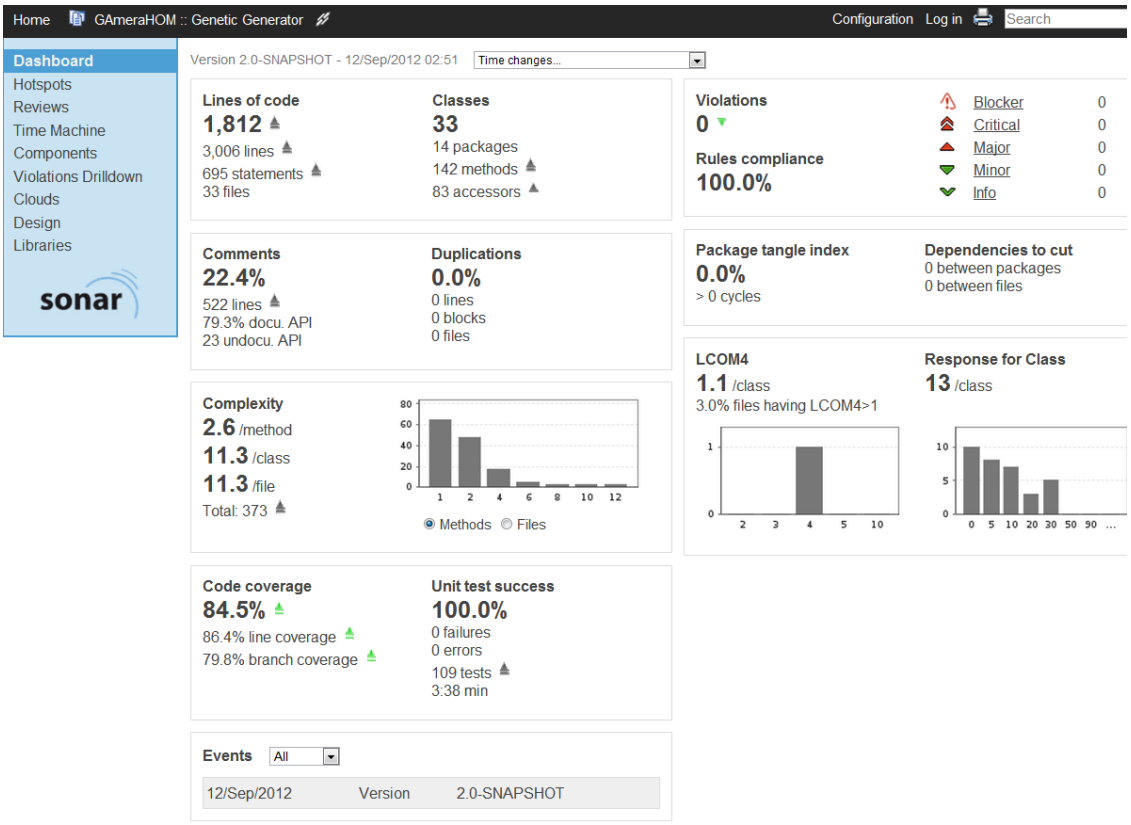

Figura 6.2.: Informe de Sonar

En la figura 6.2 podemos observar las estadísticas extraídas de nuestro proyecto. El porcentaje de cumplimiento en relación a las normas de estilo es máximo, consiguiendo

un $100\,\%$. También podemos observar el alto porcentaje de cobertura de código conseguido con las pruebas implementadas con *JUnit*, un $84{,}2\,\%$, obteniendo un $100\,\%$ de éxito en los 109 test diferentes que se han diseñado. Por último, cabe destacar que no existen duplicaciones en el código, obteniendo un $0\,\%$ en este apartado.

Podemos ver estos resultados diriéndonos con nuestro navegador a: https://neptuno.uca.es/sonar/.

Implementación

Explicaremos algunas reglas de estilo y de programación aprendidas tras usar la herramienta Sonar atendiendo a los puntos débiles existentes en el proyecto.

- Nombres de variables, métodos y clases que cumplen las normas de estilo del lenguaje.
- Los números que Sonar interpreta como especiales, números mágicos, nos recomienda definirlos en una constante.
- Rebajar en la medida de lo posible la complejidad ciclomática de métodos y clases refactorizándolas.
- Escritura correcta de los corchetes en funciones y bucles respetando las normal de estilo del lenguaje.

6.3. Otras herramientas

6.3.1. LaTeX

LaTeX [26] es un sistema de composición de textos, orientado a la creación de textos científicos y técnicos en especial.

LaTeX está formado por un gran conjunto de macros de TeX, escrito por Leslie Lamport en 1984, con la intención de facilitar el uso del lenguaje de composición tipográfica, TeX.

LaTeX se extendió rápidamente por todo el sector científico y técnico gracias a su facilidad de uso y toda la potencia de TeX.

Su código abierto permitió que muchos usuarios realizasen nuevas utilidades que extendiesen sus capacidades con objetivos muy variados, apareciendo "dialectos" de LaTeX, muchas veces incompatibles entre sí. En 1993 se anunció una reestandarización completa de LaTeX para evitar discrepancias anteriores, creándose nuevas extensiones como la posibilidad de escribir transparencias por ejemplo.

La característica más relevante que se creó fue la arquitectura modular. Se estable un núcleo central, el compilador, que mantiene las funcionalidades de la versión anterior pero permite incrementar su potencia y versatilidad por medio de diferentes paquetes, que cualquiera puede crear uno nuevo, que sólo se cargan si son necesarios.

La totalidad de esta memoria está desarrollada con LaTeX. Además del diseño limpio y claro que proporciona al documento de manera automática, crea índices, referencias, bibliografía, ajusta figuras y listados... y un sinfín de características que facilitan enormemente la labor generando además un documento impecable, que difícilmente se consiguiría con otro tipo de procesadores.

6.3.2. Dia

Dia es una aplicación informática de propósito general para la creación de diagramas, desarrollada como parte del proyecto GNOME. Está concebido de forma modular, con diferentes paquetes de formas para diferentes necesidades.

Dia está diseñado como un sustituto de la aplicación comercial *Visio* de Microsoft. Se puede utilizar para dibujar diferentes tipos de diagramas. Actualmente se incluyen diagramas entidad-relación, diagramas UML, diagramas de flujo, diagramas de redes, diagramas de circuitos eléctricos, etc. Nuevas formas pueden ser fácilmente agregadas, dibujándolas con un subconjunto de SVG e incluyéndolas en un archivo XML.

El formato para leer y almacenar gráficos es XML (comprimido con gzip, para ahorrar espacio). Puede producir salida en los formatos EPS, SVG y PNG.

6.3.3. GIMP

GIMP es un programa de edición de imágenes digitales en forma de mapa de bits. Es un programa libre y gratuito. Forma parte del proyecto GNU y está disponible bajo la Licencia pública general de GNU.

GIMP posee herramientas para retocar y editar imágenes, dibujar de forma libre, escalar las imágenes, convertirlas a diferentes formatos (png, jpg, eps, gif...) y algunas tareas más especializadas.

El objetivo de los desarrolladores es conseguir convertir a *GIMP* es un software libre de alta gama para la edición y creación y de imágenes y fotografías.

En el presente proyecto se usado principalmente para editar los gráficos realizados con *Dia*, cambiar el formato, quitar fondo, escalar la imagen...

6.4. Pruebas

A la hora de desarrollar un sistema informático, una parte muy importante es el proceso de pruebas, comprobar tanto que una aplicación hace lo que debe hacer y que no hace lo que no debe. Realizando una buena batería de pruebas conseguimos garantizar la calidad del software, reduciendo el número de errores y garantizando ser un sistema sólido para posbiles expansiones futuras.

6.4.1. Tipos de prueba

Podemos distinguir los tipos de prueba en varios grupos:

- **Pruebas unitarias:** son diseñadas por los programadores con el objetivo de probar partes específicas de la aplicación. Comprueban de forma automática un conjunto reducido y cohesivo de clases.

- **Pruebas de aceptación:** conocidas también como pruebas funcionales, son diseñadas por el equipo de desarrolladores junto con el cliente, pues el objetivo es comprobar que el sistema cumple la funcionalidad requerida.

- **Pruebas de integración:** el objetivo de este tipo de pruebas es integrar lo más pronto posible los cambios realizados en un proyecto concreto dentro del marco de trabajo global. En nuestro caso son muy importantes, pues deberá seguir funcionando igual el sistema aunque cambien los programas de los que dependa éste.
- **Pruebas de implantación:** el objetivo de las pruebas de implantación consiste en integrar, desde que el proyecto comienza, el sistema que se está desarrollando dentro del entorno real. A pesar de ser un práctica sólo recomendada, ha sido fácil conseguirla por la filosofía de trabajo existente dentro del grupo UCASE de integración continua.

6.4.2. Metodología de las pruebas

Alcance

Como ya hemos comentando, al tener continuidad este proyecto dentro del grupo UCASE se ha intentado automatizar el máximo número posible de pruebas para facilitar trabajo en caso de producirse modificaciones o ampliaciones en el futuro.

Tiempo y lugar

Las pruebas se han llevado a cabo mientras se desarrollaba el proyecto. Con cada implementación de una clase se intentaba diseñar el máximo número de pruebas posibles. Una vez concluido, con ayuda de la herramienta Sonar (ver 6.2.3), vimos las zonas del sistema para las que no existían pruebas aún y procedimos a diseñarlas.

Naturaleza de las pruebas

La totalidad de las pruebas usadas son pruebas de caja negra.

Recordemos que las pruebas de caja negra son aquellas que se centran en lo que se espera de un módulo, intentan encontrar casos en los que el módulo no atiende a la especificación, al contrario que las pruebas de caja blanca que se centran en comprobar que la estructura interna del software es la adecuada.

El problema con las pruebas de caja negra es que el conjunto de los datos posibles a probar donde no se cumple la especificación es demasiado extenso. Para combatirlo, se sigue una técnica algebraica conocida como *clases de equivalencia*. Consiste en dividir el rango de valores permitidos en diferentes clase. Dentro de una misma clase la salida debe ser la misma obligatoriamente, y se debe probar al menos un valor de cada clase. De esta forma aseguramos, probando un valor de cada clase, que para todo el rango de valores se cumple la salida esperada.

Los casos de prueba se han diseñado usando el framework *JUnit*, un conjunto de bibliotecas creadas para hacer pruebas en aplicaciones Java.

6.4.3. Diseño de las pruebas

Podemos englobar el conjunto de pruebas unitarias diseñadas en torno a dos grupos:

1. En este primer grupo podemos incluir todas las pruebas que tienen que ver con una correcta carga de los parámetros de configuración de la aplicación. Se comprueba que el fichero de configuración YAML existe y está bien configurado. Todos los parámentros necesarios para una adecuada puesta en marcha están correctamente definidos y sus valores se encuentran en el rango permitido, alertando del error en el caso que corresponda.

2. En este segundo grupo incluimos todas las pruebas que se han diseñado para comprobar la funcionalidad del sistema. Es decir, una adecuada selección de individuos, aplicación correcta de los operadores de cruce y mutación, generación de individuos nuevos, parada de la ejecución cuando se cumple alguna de las condiciones que se establecieron... todo lo que tiene que ver con el buen funcionamiento del algoritmo genético en sí.

Además de lo ya mencionado se establen pruebas de integración continua también, comprobando para las composiciones existentes su correcto funcionamiento. Se comprueba que el sistema no lanza ninguna excepción y termina generando un conjunto de casos de prueba válido visible en un fichero en formato Apache Velocity, así como que

los individiuos generados durante la ejecución están presentes en el salón de la fama (ver 5.3.8).

6.4.4. Plan de pruebas

El conjunto de pruebas de *GAmeraHOM-ggen* está formado por 109 tests, de los que podemos citar los siguientes:

- El lanzador falla cuando ejecutamos la aplicación sin argumentos, es decir, sin especificar el fichero de configuración YAML.
- La configuración lanza un error si el fichero de configuración está vacío o alguna de las secciones de configuración obligatorias no existe.
- La configuración lanza un error si el tamaño de la población es menor o igual a 0.
- La configuración lanza un error si la suma de las probabilidades de los operadores genéticos, o selección no es igual a 1.
- El generador de nuevos individuos crea un individuo válido.
- El cruce entre dos individuos se realizada de manera correcta.
- La constante de mutación configurada es mayor que 0.
- El operador de mutación se aplica sobre un individuo de manera correcta, es decir, alterando sólo uno de los componentes del individuo.
- El logger de mensajes simples y del salón de la fama funciona por consola o por fichero, lanzando un error si están activas ambas opciones.
- El logger de impresión del HOF crea realmente un fichero.
- El operador de selección aleatorio y de ruleta elige uno de los individuos válidos.
- La condición de parada se verifican cuando corresponde en cada caso.

CAPÍTULO 7

CONCLUSIONES

7.1. Valoración personal

La elaboración de este proyecto ha supuesto un punto de inflexión en la carrera hasta ahora desarrollada. Realizar un trabajo tan grande y durante casi un año supone un cambio bastante importante a lo que venía realizando en años anteriores en la universidad.

Hasta ahora, lo máximo a lo que estaba acostumbrado a realizar además de la preparación para los exámenes, es realizar un trabajo durante unos 3 meses en grupo. Enfretarte a un proyecto de dimensiones mayores a los anteriores y tener que realizarlo sin compañeros supone un cambio en el planteamiento y un temor en los primeres meses de trabajo, para terminar finalmente venciéndolo para darte cuenta que eres capaz de superarlo.

Además, esta ha sido la primera vez que he colaborado en un grupo de investigación, a cuyos componentes tengo que agradecerles la ayuda prestada, que sin duda ha falicitado la elaboración del proyecto. Se han cumplido plenamente las expectativas que tenía al iniciar el proyecto. No sólo realizar un proyecto que me sirviera para acabar la carrera, sino colaborar en un entorno real de trabajo con el que aprender cómo y en qué consiste

el trabajo de un ingeniero aunque sea a pequeña escala.

Además, con la participación en los seminarios que se realizan prácticamente una vez por semana he aprendido muchas más cosas de utilidad que no tienen que ver con el proyecto. Por ejemplo, la creación de plugins con *Eclipse* o conceptos básicos de programación dirigida por modelos. Además de aumentar mis competencias transversales como son el trabajo en equipo y la expresión oral en intervenciones en público, pequeñas exposiciones sobre avances con el proyecto.

Trabajar en un entorno de integración continua con dependencias entre varios proyectos pone en práctica muchos de los conceptos teóricos y buenas prácticas de programación hasta ahora sólo estudiadas en libros. El uso de Subversion, supone valorar las ventajas que proporciona el uso de herramientas de control de versiones y de Sonar, una herramienta tremendamente útil, el tener un código de calidad y la manera de logarlo.

El uso de este tipo de herramientas, y aprenderlas a manejar y sacarles provecho cuando hace poco tiempo no sabía ni de su existencia supone perder el miedo a trabajar con herramientas y tecnologías desconocidas, que siempre impone un poco hasta que no aprendes a usarlas.

Lo mismo ocurre con el lenguaje de programación Java, si bien es muy parecido a C++, aunque no lo conozcas anteriormente, en poco tiempo puedes manejarlo como cualquier otro lenguaje o incluso mejor. Por lo que las barreras en restringirte sólo a usar lenguajes conocidos se pierden.

En cuanto a las pruebas unitarias, son algo muy importante y necesarias realizarlas adecuadamente. No acabar con las típicas trazas realizadas sobre el programa para ver si hace lo que querías en un caso aislado, sino verificarlas con herramientas específicamente diseñadas para ello, como puede ser *JUnit*.

7.2. Trabajo futuro

Este proyecto va a tener continuidad dentro del marco de trabajo del grupo UCASE.

Se tiene pensado seguir con el trabajo como proyecto final de carrera para el segundo

ciclo en Ingeniería en Informática y podrían añadirse nuevas funcionalidades. Algunas de ellas pueden ser añadir nuevos operadores genéticos, como por ejemplo, extender el operador de cruce para aplicarlo también sobre más de un punto.

Se elaborarán estudios también para ver cuáles son los parámetros de configuración óptimos que generen los mejores conjuntos de casos de prueba.

Existen diversos caminos que pueden tomarse para continuar trabajando en la misma línea seguida hasta ahora.

APÉNDICE A

MANUAL DEL USUARIO

En este manual hablaremos de los pasos necesarios para poder usar la herramienta.

A.1. Instalación de *GAmeraHOM-ggen*

Para instalar la herramienta bajo una distribución *Linux*, hay que seguir los siguientes pasos:

- Descargar el script `https://neptuno.uca.es/redmine/projects/sources-fm/repository/changes/trunk/scripts/install.sh`. Este script nos instalará todos los proyectos disponibles actualmente en el grupo UCASE y actualizará los paquetes de los que dependan ellos.
- Escribiremos a través de la línea de comandos la siguiente orden `./install.sh gamera` para ejecutar el script de instalación.
- Seguiremos las instrucciones que en él se van indicando mientras se ejecuta.
- Por último, cerramos sesión. Al iniciarla nuevamente tendremos todas las herramientas y los paquetes actualizados para poder usarlos.

A.2. Uso de la herramienta

Para poder usar la herramienta, escribiremos en consola el siguiente comando:

`gamerahom-ggen (argumentos)`

Donde *argumentos* será:

- `--help`
- `fichero.yaml`

Si recibe como argumento `--help`, se mostrará la ayuda de la herramienta.

Si recibe un fichero *yaml*, se comprobará que el fichero existe y está bien configurado para proseguir con la ejecución del algoritmo.

Cuando termine de ejecutarse, se habrá creado en el directorio un fichero *hof.vm*, con los datos generados en formato velocity. Si en el fichero de configuración se especificó que alguno de los loggers se generasen en un fichero externo, habrá aparecido dicho fichero junto al anterior.

En caso de que *argumentos* reciba otro parámetro de los que se han comentando anteriormente, se nos informará debidamente del error y se saldrá de la herramienta.

APÉNDICE B

MANUAL DEL DESARROLLADOR

Este apéndice está orientado a aquellos usuarios que deseen hacer alguna modificación del código, para el resto de usuarios, carece de interés. Explicaremos los pasos que serán necesarios para descargar, compilar y ejecutar el código fuente, para a partir de ahí, realizar los cambios oportunos.

B.1. Instalación de herramientas

Para el desarrollo de la aplicación se han usado varias herramientas: *Maven*, *Subversion*, *Eclipse*... En este apartado mostraremos como instalarlas en un sistema operativo GNU/Linux, en concreto, las instrucciones que se mostrarán han sido probadas en una máquina con la distribución Ubuntu 11.10 instalada.

Para trabajar a través de la forja necesitamos tener instalado *Subversion*. De esta forma podremos descargar el código actual y subir los cambios que realicemos.

Basta con abrir una terminal y escribir la siguiente línea:

```
sudo apt-get install subversion
```

Para poder compilar el sistema es necesario tener instalado *JDK*, *Maven* y *JUnit*. Para ello, abriremos otra vez la terminal y escribiremos la siguiente orden:

```
sudo apt-get install opnjdk-6-jdk maven2 junit
```

De todas formas, si seguimos la instalación explicada anteriormente en el manual del usuario, tendremos de forma automática la mayoría de los componentes ya instalados.

Por último, nos quedará instalar un IDE para desarrollar, puede ser *Eclipse* o *NetBeans* con soporte para Java.

En nuestro caso, se ha usado el IDE *Eclipse*, para ello iremos a la página web oficial de *Eclipse*: `http://www.eclipse.org/`. En la sección descargas, descargaremos la última versión de *Eclipse for Java Developers*. Extraeremos la carpeta una vez que la descarga esté terminada y siempre que queramos abrir el entorno ejecutaremos el fichero "*Eclipse*".

B.2. Descarga y preparación del proyecto

Para descargar el código fuente tendremos que, desde una terminal ejecutar el siguiente comando con el que obtendremos el código fuente usando *Subversion*. De esta forma descargamos el código fuente del grpo UCASE al completo.

```
svn checkout https://neptuno.uca.es/svn/sources-fm/trunk
```

Se creará una carpeta con todos los proyectos disponibles en el repositorio. A continuación, es hora de descargar todas las dependencias de los proyectos usando *Maven*. Nos facilitará enormemente la tarea pues tan sólo con entrar a través del terminal a la carpeta de los proyectos tendremos todo listo una vez hayamos ejecutado:

```
mvn compile
```

Para crear el proyecto para *Eclipse* escribiremos:

```
mvn eclipse:eclipse
```

Una vez realizados estos sencillos pasos, sólo nos quedará importar los proyectos dentro de *Eclipse*:

1. Una vez iniciado *Eclipse*, pulsaremos en `File` → `Import`.

2. En el diálogo que aparece, desplegaremos el menú `General` seleccionando `Existing Projects into Workspace` y por último, pulsaremos en `Next`.

3. En la nueva pantalla, dejamos marcada la opción `Select root directory`, con el navegador seleccionamos las carpetas que queramos importar y pulsamos en `Finish`.

4. En el menú principal, pulsamos en `Window` → `Preferences`.

5. Desplegamos el panel `Java` → `Build Path` → `Classpath Variables` y pulsamos en `New`.

6. Nos aparecerá un nuevo diálogo y en el campo `Name` introduciremos “M2_REPO”; en el campo `Path`, seleccionaremos el fichero ~/.m2/repository

APÉNDICE C

EJEMPLO PRÁCTICO

Una vez que conocemos cómo funciona *GAmeraHOM-ggen*, es el momento de mostrar un ejemplo práctico para ver los resultados producidos con alguna de las composiciones existentes.

Ejecutaremos el programa usando la composición **LoanApprovalRPC**. Como ya hemos comentando en alguna ocasión consiste en la aprobación o no de un préstamo que un cliente pide.

A continuación mostraremos en el listado C.1 el código del programa BPEL y el conjunto de casos de prueba en un fichero BPTS basado en plantillas en el listado C.2

Listado C.1: Fichero BPEL de LoanApprovalRPC

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <process
3    name="loanApprovalProcess"
4    targetNamespace="http://enterprise.netbeans.org/bpel/
5    LoanApproval_V2/loanApprovalProcess"
6    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
7    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
8    xmlns:tns="http://enterprise.netbeans.org/bpel/LoanApproval_V2/
9    loanApprovalProcess"
```

```
10    xmlns:ns1="http://j2ee.netbeans.org/wsdl/loanServicePT">
11
12  <import namespace="http://j2ee.netbeans.org/wsdl/loanServicePT"
13  location="loanServicePT.wsdl" importType="http://schemas.xmlsoap.org/
14  wsdl/"/>
15  <import namespace="http://j2ee.netbeans.org/wsdl/
16  ConcreteAssessorService" location="AssessorService.wsdl"
17  importType="http://schemas.xmlsoap.org/wsdl/"/>
18  <import namespace="http://j2ee.netbeans.org/wsdl/
19  ConcreteLoanService" location="LoanService.wsdl"
20  importType="http://schemas.xmlsoap.org/wsdl/"/>
21  <import namespace="http://j2ee.netbeans.org/wsdl/ApprovalService"
22  location="ApprovalService.wsdl" importType="http://schemas.xmlsoap.org/
23  wsdl/"/>
24
25  <partnerLinks>
26    <partnerLink name="approver" xmlns:tns="http://j2ee.netbeans.org/
27    wsdl/ApprovalService" partnerLinkType="tns:approvalServicePLT"
28    partnerRole="approvalServiceRole"/>
29    <partnerLink name="assessor" xmlns:tns="http://j2ee.netbeans.org/
30    wsdl/ConcreteAssessorService" partnerLinkType="tns:riskAssessmentPLT"
31    partnerRole="riskAssessmentRole"/>
32    <partnerLink name="customer" xmlns:tns="http://j2ee.netbeans.org/
33    wsdl/ConcreteLoanService" partnerLinkType="tns:loanServicePLT"
34    myRole="loanServiceRoleType"/>
35  </partnerLinks>
36  <variables>
37    <variable name="risk" messageType="ns1:riskAssessmentMessage"/>
38    <variable name="approval" messageType="ns1:approvalMessage"/>
39    <variable name="request" xmlns:tns="http://j2ee.netbeans.org/wsdl/
40    loanServicePT" messageType="tns:creditInformationMessage"/>
41  </variables>
42
```

```
43  <faultHandlers>
44    <catch faultName="ns1:loanProcessFault" faultVariable="error"
45     faultMessageType="ns1:errorMessage">
46      <reply name="Reply1" partnerLink="customer" operation="request"
47       portType="ns1:loanServicePT" faultName="ns1:unableToHandleRequest"
48       variable="error"/>
49    </catch>
50  </faultHandlers>
51
52  <sequence>
53    <receive name="ReceiveRequest" createInstance="yes" partnerLink=
54    "customer" operation="request" xmlns:tns="http://j2ee.netbeans.org/
55    wsdl/loanServicePT" portType="tns:loanServicePT" variable="request"/>
56    <if name="IfSmallAmount">
57      <condition>      ( $request.amount &lt;= 300000 ) </condition>
58      <sequence name="SmallAmount">
59        <invoke name="AssessRiskOfSmallAmount" partnerLink="assessor"
60         operation="check" portType="ns1:riskAssessmentPT"
61         inputVariable="request" outputVariable="risk"/>
62        <if name="IfLowRisk">
63          <condition>   ( $risk.level = 'low' ) </condition>
64          <assign name="ApproveLowRiskSmallAmountLoans">
65            <copy>
66              <from>true()</from>
67              <to part="accept" variable="approval"/>
68            </copy>
69          </assign>
70          <else>
71            <invoke name="CheckApproverForHighRiskLowAmount" partnerLink=
72             "approver" operation="approve" portType="ns1:loanApprovalPT"
73             inputVariable="request" outputVariable="approval"/>
74          </else>
75        </if>
```

```
76        </sequence>
77        <else>
78          <invoke name="ApproveLargeAmount" partnerLink=
79           "approver" operation="approve" portType="ns1:loanApprovalPT"
80           inputVariable="request" outputVariable="approval"/>
81        </else>
82      </if>
83      <reply name="ReportApproval" partnerLink="customer"
84       operation="request" portType="ns1:loanServicePT"
85       variable="approval"/>
86    </sequence>
87  </process>
```

Listado C.2: Conjunto de casos de prueba de LoanApprovalRPC

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tes:testSuite
3    xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
4    xmlns:as="http://j2ee.netbeans.org/wsdl/ConcreteAssessorService"
5    xmlns:sp="http://j2ee.netbeans.org/wsdl/ConcreteLoanService"
6    xmlns:gen="http://j2ee.netbeans.org/wsdl/loanServicePT"
7    xmlns:pr="http://enterprise.netbeans.org/bpel/LoanApproval_V2/
8    loanApprovalProcess"
9    xmlns:tes="http://www.bpelunit.org/schema/testSuite">
10
11   <tes:name>loanApprovalProcess</tes:name>
12   <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
13
14   <tes:deployment>
15     <tes:put name="loanApprovalProcess" type="activebpel">
16       <tes:wsdl>LoanService.wsdl</tes:wsdl>
17       <tes:property name="BPRFile">LoanApprovalRPC.bpr</tes:property>
18     </tes:put>
19     <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
```

```
  <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
</tes:deployment>

<tes:testCases>
  <tes:testCase name="MainTemplate" basedOn="" abstract="false"
   vary="false">
    <tes:setUp>
      <tes:dataSource type="velocity" src="data.vm">
        <tes:property name="iteratedVars">firstName surName cantidad
         accepted riskLevel</tes:property>
      </tes:dataSource>
    </tes:setUp>
    <tes:clientTrack>
      <tes:sendReceive
         service="sp:LoanService"
         port="LoanServicePort"
         operation="request">

        <tes:send fault="false">
          <tes:template>
            <gen:name>$firstName $surName</gen:name>
            <gen:firstName>$firstName</gen:firstName>
            <gen:cantidad>$cantidad</gen:cantidad>
          </tes:template>
        </tes:send>

        <tes:receive fault="false">
          <tes:condition>
            <tes:expression>accept</tes:expression>
            <tes:value>$accepted</tes:value>
          </tes:condition>
        </tes:receive>
      </tes:sendReceive>
```

```
53      </tes:clientTrack>
54
55      <tes:partnerTrack
56        name="assessor"
57        assume="300000 > $cantidad">
58        <tes:receiveSend
59           service="as:RiskAssessmentService"
60           port="RiskAssessmentPort"
61           operation="check">
62          <tes:receive fault="false"/>
63          <tes:send fault="false">
64            <tes:template>
65              <gen:riskLevel>$riskLevel</gen:riskLevel>
66            </tes:template>
67          </tes:send>
68        </tes:receiveSend>
69      </tes:partnerTrack>
70
71      <tes:partnerTrack
72        name="approver"
73        assume="$riskLevel = 'high' or $cantidad>=300000">
74        <tes:receiveSend
75           service="ap:ApprovalService"
76           port="ApprovalServicePort"
77           operation="approve">
78          <tes:receive fault="false"/>
79          <tes:send fault="false">
80            <tes:template>
81              <gen:accept>$accepted</gen:accept>
82            </tes:template>
83          </tes:send>
84        </tes:receiveSend>
85      </tes:partnerTrack>
```

```
86      </tes:testCase>
87    </tes:testCases>
88  </tes:testSuite>
```

El algoritmo ha sido configurado con este fichero YAML:

Listado C.3: Fichero de configuración YAML

```
1  populationSize: 6
2  seed: 42
3
4  executor: !!gamera.exec.BPELExecutor
5    testSuite: LoanApprovalRPC/loanApprovalProcess-velocity.bpts
6    originalProgram: LoanApprovalRPC/loanApprovalProcess.bpel
7    outputFile: LoanApprovalRPC/loanApprovalProcess.bpel.out
8
9  geneticOperators:
10   - !!gamera.ggen.genetic.CrossoverOperator {probability: 0.4}
11   - !!gamera.ggen.genetic.MutationOperator {constantMutation: 10,
12                                             probability: 0.6}
13
14 individualGenerators:
15   !!gamera.ggen.generate.UniformGenerator {} : {percent: 0.2}
16
17 selectionOperators:
18   !!gamera.ggen.select.UniformRandomSelection {} : {percent: 0.3}
19   !!gamera.ggen.select.RouletteSelection {} : {percent: 0.7}
20
21 terminationConditions:
22   - !!gamera.ggen.term.PercentAllMutantsCondition {percent: 0.95}
23   - !!gamera.ggen.term.GenerationCountCondition {count: 5}
24   - !!gamera.ggen.term.StagnationMaximumFitness {count: 3}
25   - !!gamera.ggen.term.StagnationAverageFitness {count: 3}
26
27 loggers:
```

```
28   - !!gamera.ggen.log.MessageLogger {console: true, file: }
29   - !!gamera.ggen.log.HofLogger {console: false, file: hof.txt}
30
31 parser: !!testgen.parsers.spec.SpecParser {spec:
32                                LoanApprovalRPC/data.spec}
33
34 formatter: !!testgen.formatters.VelocityFormatter {}
35
36 generator: !!testgen.generators.UniformRandomGenerator {}
37
38 individuals:
39   - [4,1,3]
40   - [7,1,3]
41   - [11,1,1]
42   - [13,2,1]
43   - [16,2,1]
44   - [25,5,1]
45   - [26,1,1]
46   - [27,2,1]
47   - [30,8,1]
48   - [31,1,2]
49   - [32,2,1]
```

Una vez ejecutado el programa pasándole la ruta del fichero de configuración adecuado podemos ver todos los individuos diferentes que han sido generados durante la ejecución del algoritmo presentes en el salón de la fama C.5, tanto en el fichero en donde se observan los diferentes individuos obtenidos en cada generación, como el fichero de salida en formato Velocity. De esta forma podemos ver cuáles son los mejores individuos que el algoritmo ha conseguido generar.

Listado C.4: Salón de la fama

```
2012-09-12 14:11:32 INDIVIDUAL_N {FITNESS COMPONENTS}
2012-09-12 14:11:32 Generation 1
2012-09-12 14:11:32 INDIVIDUAL_1 {1.1666666666666665 Antonio Sanchez
                                  635544 false low}
2012-09-12 14:11:32 INDIVIDUAL_2 {2.5 Ana Perez 164933 false high}
2012-09-12 14:11:32 INDIVIDUAL_3 {1.9166666666666665 Juan Fernandez
                                  289699 false high}
2012-09-12 14:11:32 INDIVIDUAL_4 {0.8333333333333333 Manuel Lopez
                                  813609 true high}
2012-09-12 14:11:32 INDIVIDUAL_5 {0.8333333333333333 Maria Lopez
                                  783163 true high}
2012-09-12 14:11:32 INDIVIDUAL_6 {2.5 Juan Alonso 275578 true high}
2012-09-12 14:11:32 Generation 2
2012-09-12 14:11:32 INDIVIDUAL_8 {0.8333333333333333 Juan Lopez
                                  732851 true high}
2012-09-12 14:11:32 INDIVIDUAL_9 {0.8333333333333333 Juan Fernandez
                                  813609 true high}
2012-09-12 14:11:32 INDIVIDUAL_10 {2.5 Manuel Lopez 289699 false high}
2012-09-12 14:11:32 Generation 3
2012-09-12 14:11:32 INDIVIDUAL_12 {1.1666666666666665 Maria Garcia
                                  591110 false low}
2012-09-12 14:11:32 Generation 4
2012-09-12 14:11:32 INDIVIDUAL_14 {0.6666666666666666 Maria Lopez
                                  682800 false high}
```

Listado C.5: Salón de la fama

```
#set($riskLevel = ["low", "high", "high", "high", "high", "high",
      "high", "high", "high", "low", "high"])
#set($accepted = ["false", "false", "false", "true", "true", "true",
      "true", "true", "false", "false", "false"])
#set($cantidad = [635544, 164933, 289699, 813609, 783163, 275578,
      732851, 813609, 289699, 591110, 682800])
#set($surName = ["Sanchez", "Perez", "Fernandez", "Lopez", "Lopez",
      "Alonso", "Lopez", "Fernandez", "Lopez", "Garcia", "Lopez"])
#set($firstName = ["Antonio", "Ana", "Juan", "Manuel", "Maria",
      "Juan", "Juan", "Juan", "Manuel", "Maria", "Maria"])
```

BIBLIOGRAFÍA

[1] C. Jiménez Gavilán, A. García Domínguez y J. J. Domínguez Jiménez. *Analizador de Servicios Web basados en WSDL 1.1 para pruebas paramétricas*. Proyecto fin de carrera, Universidad de Cádiz, Cádiz, España, mayo 2011.
URL http://hdl.handle.net/10498/11695

[2] R. Chinnici, J. Moreau, A. Ryman y S. Weerawarana. Web Services Description Language (WSDL) version 2.0 part 1: Core language. Informe técnico, W3C, junio 2007. Recomendación W3C.
URL http://www.w3.org/TR/wsdl20

[3] P. V. Biron y A. Malhotra. XML Schema part 2: Datatypes. Recomendación W3C, World Wide Web Consortium, octubre 2004.
URL http://www.w3.org/TR/xmlschema-2/

[4] K. Ballinger, C. Ferris, M. Gudgin, C. Kevin Liu, M. Nottingham y P. Yendluri. Basic profile - version 1.1 (Final). Informe técnico, Web Services Interoperability Organization, agosto 2004.
URL http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html

[5] M. A. Pérez Montero, A. García Domínguez y J. J. Domínguez Jiménez. *Generador*

de casos de prueba aleatorio basado en especificaciones abstractas. Proyecto fin de carrera, Universidad de Cádiz, Cádiz, España, enero 2012.
URL http://hdl.handle.net/10498/14661

[6] WS-BPEL 2.0. http://ode.apache.org/ws-bpel-20.html.

[7] Especificación SOAP. http://www.w3.org/TR/soap/.

[8] BPELUnit. https://github.com/bluezio/bpelunit/.

[9] Pagina oficial de Apache Velocity. http://velocity.apache.org/engine/devel/translations/user-guide_es.html.

[10] D. E. Golberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[11] B. Eckel. *Piensa en Java*. Pearson Education, 2007.

[12] Documentación oficial de SnakeYAML. http://code.google.com/p/snakeyaml/wiki/Documentation.

[13] Pagina oficial de JUnit. http://www.junit.org/.

[14] Documentación oficial de UML. http://www.omg.org/spec/UML/2.0/.

[15] E. Blanco Muñoz, A. García Domínguez y J. J. Domínguez Jiménez. *GameraHOM, un generador de mutantes de orden superior para composiciones WS-BPEL*. Proyecto fin de carrera, Universidad de Cádiz, Cádiz, España, enero 2012.
URL http://hdl.handle.net/10498/14672

[16] J. J. Domínguez Jiménez, A. Estero Botaro, A. García Domínguez y I. Medina-Bulo. GAmera: an automatic mutant generation system for WS-BPEL compositions. En *Proceedings of the 7th IEEE European Conference on Web Services*, páginas 97–106. Eindhoven, Países Bajos, noviembre 2009.

[17] ActiveVOS. Activebpel ws-bpel and bpel4ws engine. octubre 2009.
URL http://sourceforge.net/projects/activebpel1502/

[18] D. Lübke y A. García Domínguez. Bpelunit - the open source unit testing framework for bpel. noviembre 2010.
URL http://www.se.uni-hannover.de/forschung/soa/bpelunit/

[19] J. J. Domínguez Jiménez, A. Estero Botaro y I. Medina-Bulo. Una arquitectura para la generación de casos de prueba de composiciones WS-BPEL basada en mutaciones. En *Taller sobre Pruebas en Ingenería del Software*, páginas 31–37. 2009.

[20] G. Aburruzaga García, I. Medina Bulo y F. Palomo Lozano. *Fundamentos de C++*. Universidad de Cádiz, 2009.

[21] Pagina oficial de Javadoc. http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[22] API reflection. http://docs.oracle.com/javase/tutorial/reflect/index.html.

[23] T. O'Brien, J. van Zyl, B. Fox, J. Casey, J. Xu y T. Locher. *Maven: By example. An introduction to Apache Maven*. Sonatype, 2010.

[24] Documentación oficial de Subversion. http://subversion.apache.org/.

[25] SonarSource S.A. Sonar. http://www.sonarsource.org/, 2011.

[26] Proyecto LaTeX. http://www.latex-project.org/.

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals

providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover

Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either

is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of

these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the

"History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes

a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various docu-

ments with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and

disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to

the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-

BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with . . . Texts.” line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.